

Sealable Metaobjects for Common Lisp

Marco Heisig
FAU Erlangen-Nürnberg
Cauerstraße 11
Erlangen 91058, Germany
marco.heisig@fau.de

ABSTRACT

We introduce the concept of sealable metaobjects, i.e., classes, generic functions, and methods, whose behavior is restricted to allow for some static analysis. Then we use these sealable metaobjects to define fast generic functions, a variant of standard generic functions that allow for call site optimization — ranging from faster method dispatch to inlining of entire effective methods. Fast generic functions support almost all features of standard generic functions, including custom method combinations and non-trivial references to the next method. Our benchmarks show that a straightforward implementation of Common Lisp’s sequence functions using these fast generic functions is competitive with the corresponding built-in sequence functions of SBCL. Fast generic functions are thus an attractive drop-in replacement for standard generic functions in performance critical codes.

KEYWORDS

Common Lisp, Metaobject Sealing, Generic Function Inlining, CLOS

ACM Reference format:

Marco Heisig. 2020. Sealable Metaobjects for Common Lisp. In *Proceedings of the 13th European Lisp Symposium, Zürich, Switzerland, April 27–28 2020 (ELS’20)*, 7 pages.
DOI: 10.5281/zenodo.3743823

1 INTRODUCTION

After three decades, the Metaobject Protocol[7] of the Common Lisp Object System is still the pinnacle of object oriented programming technology. Objects, functions and classes can be created, modified and augmented continuously, without sacrificing either correctness¹ or performance. Because generic functions, methods and classes are themselves objects, they can also be extended. This capability allows programmers to explore a whole space of object oriented paradigms.

Yet there is one rather fundamental limitation of CLOS. Since objects can be redefined at any time, implementations are forced to use at least one indirection when calling a generic function. It is not possible to move any part of a generic function into the call site. This indirection is great for modularity and extensibility, but means

that generic functions are inherently ill suited for the representation very lightweight operations such as arithmetic operations on floating-point numbers. In particular, new Lisp programmers are frequently disappointed that they cannot extend standard functions like `+` and `find` to new data types.

In this paper, we show how these limitations can be lifted while preserving most of the flexibility that programmers expect from CLOS. We first define a set of sealable metaobjects with carefully chosen limitations on how they can be extended and redefined. Then, in a second step, we show how these metaobjects can be used to implement *fast generic functions*, a variant of standard generic functions that perform several powerful optimizations. Finally, we show how fast generic functions can be used to define efficient number and sequence functions and present some promising benchmark results. All our work is available on Quicklisp. The library for metaobject sealing is called `SEALABLE-METAOBJECTS`, and the library that provides fast generic functions is called `FAST-GENERIC-FUNCTIONS`.

2 RELATED WORK

The idea of sealing metaobjects to speed up certain kinds of programs is not new. Similar functionality can be found both in Common Lisp libraries, and in the Dylan programming language. Our main contribution is to gather these ideas and to make them accessible in a high quality implementation. Therefore, we are indebted to the following prior endeavors:

- Henry Baker has proposed a subset of CLOS called `STATIC CLOS`[2], where it is not possible to change the class of any object and the class hierarchy is fixed at compile-time. In doing so, Static CLOS allows the compiler to eliminate most of the overhead that is normally associated with calls to generic functions. In contrast to our technique, Static CLOS enforces its restrictions globally, not just for a chosen set of metaobjects, and it bypasses most of the machinery of the Metaobject Protocol altogether.
- The Dylan programming language[9] has a feature called *define sealed domain* that permanently freezes a part of the domain of a generic functions, while still allowing changes outside of the sealed domain. Our technique closely mimics this feature, except for the necessary adaptations to integrate it into Common Lisp.
- The library `INLINED-GENERIC-FUNCTION`[1] by Masataro Asai provides generic functions that can be inlined into the call site. Our work is directly inspired by this library, but tries to improve upon several issues. The crucial difference is that the inlined-generic-function library works by inlining the dispatch function and all reachable effective

¹Barring eccentricities like concurrent modification of the class hierarchy.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ELS’20, Zürich, Switzerland

© 2020 Copyright held by the owner/author(s).

DOI: 10.5281/zenodo.3743823

methods into the call site, while our technique will only inline effective methods if they are marked as reasonably cheap, and if the dispatch function can be reduced to a single case. By using more conservative inlining rules, we achieve that users can unconditionally use fast generic functions without worrying about code bloat.

- The library `SPECIALIZATION-STORE`[4] by Mark Cox provides an alternative to generic functions that trade various features of CLOS — like method combinations, argument precedence ordering and class-based dispatch — for better compile-time optimizations.
- The library `STATIC-DISPATCH`[6] by Alexander Gudev provides a mechanism of static dispatch that is implemented via compiler macros and shadowing of `defmethod`. However, the library is not extensible and doesn't support method combinations.

3 METAOBJECT SEALING

Common Lisp permits incremental changes to the class hierarchy and to the methods of a generic function. The purpose of metaobject sealing is to restrict these capabilities in order to enable static analysis and optimization.

3.1 Sealable Metaobjects

Sealable metaobjects are classes, generic functions, methods, slot-definitions or method combinations with two states — sealed and unsealed. Once a metaobject is sealed, it remains sealed indefinitely. Furthermore, the following restrictions apply to all sealed metaobjects:

- (1) Calling `reinitialize-instance` on a sealed metaobject has no effect.
- (2) It is an error to change the class of a sealed metaobject.
- (3) It is an error to change the class of any object to a sealed metaobject.
- (4) It is an error to change the class of an instance of a sealed metaobject.
- (5) Each superclass of a sealed metaobject must be a sealed metaobject.

Restriction (1) ensures that slots cannot be mutated without going through the corresponding accessors or protocols. The restrictions (2), (3) and (4) allow a compiler to reliably perform static type inference. The restriction (5) ensures that the behavior of a sealed class cannot be changed adding new superclasses to it, or by modifying existing superclasses.

Built-in classes, structure classes, and system classes are sealed metaobjects according to our definition. In addition, we provide a `sealable-metaobject-mixin` class that can be used as a building block for other kinds of sealable metaobjects.

3.2 Sealed Specializers

A sealed specializer is either a sealed class, or an EQL specializer whose object is an instance of a sealed class. Each sealed specializer also denotes a Common Lisp type. For a specializer that is a class, the type is the set of all objects that are of that class. For an EQL specializer, the type is the set of all objects that are EQL to the specializer's object.

3.3 Domains

A domain is the cartesian product of the types denoted by some specializers. A sealed domain is a domain whose constituting specializers are sealed. The domain of a method with n required arguments is the n -ary cartesian product of the types denoted by the method's specializers. We say a method is inside of a domain D if the method's domain is a subset of D , and outside of a domain D if the method's domain is disjoint from D .

A domain designator is either a domain, or a list of specializer designators. A specializer designator is either a specializer, or a class name, or an expression of the form `(eql X)` for some object X . Most functions that work with domains will accept arbitrary domain designators and convert them to domains in the obvious way.

3.4 Sealable Generic Functions

A sealable generic function is both a generic function and a sealable metaobject. It may contain any number of sealed domains. Initially, a sealable generic function has zero sealed domains. New sealed domains can be added by the function `seal-domain`, which takes a sealable generic function and a domain designator. The following rules apply to sealed domains of a generic function.

- (1) All sealed domains of a generic function must be disjoint.
- (2) Each method of the generic function must either be fully inside of the sealed domain, or fully outside of the sealed domain.
- (3) Each method inside of a sealed domain must be sealed, and all of its specializers must be sealed.
- (4) It is an error to add or remove methods from inside of a sealed domain.
- (5) It is an error to create a subclass of a sealed class that would violate any of the previous rules for any sealed generic function.

These rules ensure that the behavior of a sealed generic function whose arguments are provably within any of its sealed domains is fully known at compile time.

3.5 Potentially Sealable Methods

A potentially sealable method is both a method and a sealable metaobject. It is called *potentially* sealable because it can only be actually sealed if all of its specializers are sealable. The main purpose of potentially sealable methods is to use them (or a subclass thereof) as the generic function method class of a sealable generic function.

Potentially sealable methods have one additional feature in that they support method properties. Method properties are declarations that are automatically extracted from the method body and are made available with the generic function `method-properties`. The generic function `validate-method-property` checks whether a method property is valid. Figure 1 shows how one can define and use method properties. The purpose of method properties is to allow annotations in custom methods, e.g., whether the method is lightweight enough for inlining.

```

1 (defclass my-method
2   (potentially-sealable-standard-method)
3   ())
4
5 (defclass my-generic-function
6   (sealable-standard-generic-function)
7   ()
8   (:default-initargs
9    :method-class (find-class 'my-method))
10   (:metaclass funcallable-standard-class))
11
12 (defmethod validate-method-property
13   ((m my-method) (p (eql 'foo)))
14   t)
15
16 (defmethod validate-method-property
17   ((m my-method) (p (eql 'bar)))
18   t)
19
20 (defgeneric my-gf (x)
21   (:generic-function-class my-generic-function))
22
23 (defmethod my-gf ((x number))
24   (declare (method-properties foo)))
25
26 (defmethod my-gf ((x sequence))
27   (declare (method-properties bar foo)))
28
29 (mapcar #'method-properties
30   (generic-function-methods #'my-gf))
31 ;;; => ((bar foo) (foo))

```

Figure 1: Defining and using method properties.

3.6 Automatic Sealing

When a sealable metaobject is sealed, it automatically attempts to seal all its superclasses. When a sealable method is sealed, it automatically attempts to seal all its specializers. Furthermore, the function `seal-domain` seals both the generic function and all its methods that lie inside of the sealed domain. This way, users usually don't have to worry about the distinction between sealable and sealed metaobjects at all.

4 FAST GENERIC FUNCTIONS

So far, we have characterized sealable metaobjects solely by showing what they can't do. In this section, we will show how one can exploit the restrictions that we have introduced to speed up generic function calls. To do so, we define fast generic functions and fast methods as subclasses of sealable standard generic functions and potentially sealable standard methods, respectively.

Fast generic functions behave exactly like standard generic functions, except that their generic function method class is `fast-method`, and that they are sealable. The behavior of a fast method is just like that of a standard method, except that it inherits all the restriction

Type	Prototype
(eql 42)	42
(and integer (not (eql 42)))	5
(and real (not integer))	0.0

Figure 2: Static call signatures for the domain of real numbers, for methods specializing on (eql 42) integer and real.

that are imposed on sealable methods, and that all fast methods must be defined in a null lexical environment.

By enforcing that all fast methods are defined in a null lexical environment, we create several optimization opportunities. In particular, we may inline method functions directly into each effective method, and inline entire effective methods into the call site. Furthermore, knowing that a method is defined in a null lexical environment allows us to safely manipulate the method lambda. We exploit this to introduce a more efficient calling convention for method functions.

The next subsections contain a more detailed description of the individual steps towards faster generic functions.

4.1 Static Call Signatures

Whenever a sealed domain is added to a generic function, we compute its set of static call signatures. A static call signature consists of a list of types and a list of prototypes, each with one element per required argument of the generic function. The lists of types of each static call signature are disjoint and their union covers the entire sealed domain. The list of prototypes is chosen such that each prototype lies within the corresponding type, and not within the corresponding type of any other static call signature. We give an example of static call signatures in Figure 2.

Static call signatures are the backbone of fast generic functions. The compiler checks the types of all static call signatures to determine whether one of them is applicable. If so, our library can pass the prototypes of that static call signature to `compute-applicable-methods` to determine the ordered list of applicable methods. The list of applicable methods can then be used to compute an optimized effective method.

4.2 Call-site Analysis

Once a call to a fast generic function occurs in the source code, we have to check whether a static call signature of that function is applicable. One way to do so is by defining a compiler macro for the fast generic function, and using introspection to figure out the types of the arguments in the current environment. But this approach tends to work only if all required arguments of that generic function are lexical variables with explicitly declared types.

A more robust approach is to use the compiler infrastructure of each Lisp implementation. On SBCL, we generate one IR1 transformation for each static call signature, using `deftransform`.

4.3 Computing the Effective Methods

Once there is a unique applicable static call signature and a corresponding sorted list of applicable methods, we still have to generate an efficient effective method that can be called without going through the discriminating function.

Our initial attempt to do so was to modify fast generic functions such that a special initial method would first check a global variable, and if that variable is set, the initial method would return #'call-next-method instead of directly calling the next method. Then, we could invoke this next method function with the new arguments to get the desired behavior, without going through the discriminating function. However, we had to learn that accessing call-next-method in such a way disables many transformations that usually make generic functions reasonably efficient. The performance hit from losing these transformations was much larger than the cost of going through the discriminating function, so we had to abandon this approach.

Our current technique is to bypass the last stage of a generic function invocation, and to expand the result of calling compute-effective-method ourselves, using our own versions of make-method and call-method. This technique has the advantage that we can use a very efficient calling convention for method functions. Each method function receives one required argument per binding in its original lambda list, and has no keyword, optional and rest arguments. All parsing is performed by the effective method function. Another benefit of this technique is that it still allows fast generic functions to use arbitrary method combinations.

4.4 Call-site Optimization

Once we have determined that a fast generic function can be optimized at a particular call site, and have computed a suitable effective method, we still have to invoke the effective method somehow. To do so, we support three options:

- (1) Inlining of the entire effective method.
- (2) Inlining of keyword parsing only.
- (3) Only bypassing the generic function dispatch.

Option (1) can increase the size of the generated code dramatically, so it is only chosen when all applicable methods have the inlineable method property. Option (2) can be chosen when the fast generic function expects keyword arguments. It generates an inline lambda function that performs all necessary keyword parsing, and then calls a (not inlined) custom variant of the effective method with required arguments only. Option (3) is the default when none of the two previous options applies. It can still lead to faster performance than a regular generic function call because there is zero dispatch overhead, and because we generate effective methods that are faster than those of most implementations².

5 LIMITATIONS

In this section, we outline the quirks and limitations of our technique.

No Redefinability. Once a domain of a generic function has been sealed, there is no way to update any of the methods therein. This means that it is not possible to retroactively fix bugs in any sealed method. We are thinking about adding a unseal-domain function for development. But currently, the only portable way of fixing a bug in a sealed function is by loading the fixed code into a new Lisp image.

²This is not to say we are better than, say, PCL. We just exploit the additional restrictions that we have imposed on fast generic functions and methods.

```

32 (defun class-subclasses (class)
33   (when (symbolp class)
34     (setf class (find-class class)))
35   (let ((table (make-hash-table)))
36     (labels
37       ((subclasses (class)
38         (unless (gethash class table)
39           (setf (gethash class table) t)
40           (cons
41             class
42             (mapcan
43               #'subclasses
44               (closer-mop:class-direct-subclasses
45                 class))))))
46       (subclasses class))))
47
48 (defmacro replicate-for-each-vectoroid
49   (symbol &body body)
50   `(progn
51     ,@(loop for class in (class-subclasses 'vector)
52       unless (subtypep class '(array nil))
53         append (subst class symbol body))))
54
55 (replicate-for-each-vectoroid #1=#:vectoroid
56   (defmethod first-elt ((vector #1#))
57     (elt vector 0)))

```

Figure 3: An example of specializing on various subclasses of vector.

No Inlining of Discriminating Functions. We deliberately do not inline discriminating functions, primarily to avoid code bloat. This means that users have to make sure that the compiler can statically compute the list of applicable methods for all relevant cases. To compute this list, the compiler has to be able to distinguish whether any sealed method is applicable or not. This can be problematic when a method has EQL specializers or very narrow types (If the compiler cannot prove that an argument is either matches that specialization or doesn't match that specialization, there will be no optimization at all). For example, users should not specialize the same argument both on list and nil, or the compiler cannot optimize calls to objects that might be either conses or NIL.

Non-Portable Specializers. Specializers are either classes or EQL specializers, not types. This is problematic in our case, since the Common Lisp standard doesn't mandate that each primitive type has a corresponding class. An example is the type single-float, where the standard only guarantees the existence of a float class. Another desirable case is that of specializing on subtypes of vector, which would allow for dispatching on the element type. The good news is that doing so works in practice, and can even be made portable by using a macro that replicates a method template for all subclasses of a supplied class. The list of subclasses can be obtained using CLOSER-MOP[3]. An example of such a macro is given in Figure 3.

6 EXAMPLES

In this section, we show how fast generic functions can be used in practice.

6.1 Extensible Number Functions

This first example is about using fast generic functions to define an extensible version of the function `+`. To do so, we first define a single fast generic function for two argument addition³ in Figure 4.

```
1 (defgeneric binary-+ (x y)
2   (:generic-function-class fast-generic-function))
```

Figure 4: A generic function for two-argument addition.

With this definition in place, we can add individual methods. Initially, a single method like the one shown in Figure 5 will suffice. More methods would be needed if `cl:+` weren't so flexible already.

```
3 (defmethod binary-+ ((x number) (y number))
4   (+ x y))
```

Figure 5: A method for two-argument addition of numbers.

At this point, neither the generic function `binary-+`, nor its methods are sealed. They can be treated just like standard generic functions and methods. To enable optimization, we have to seal a domain by calling `(seal-domain #'binary-+ '(number number))`. The generic function `seal-domain` will automatically seal the generic function and all methods within the sealed domain. It would also signal an error if some methods were only partially within the sealed domain. But the most important step is that it installs a compiler transformations for calls whose arguments are provably within the sealed domain, and which provably lead to a fixed list of applicable methods.

Before we actually use our new addition function, we define an auxiliary function `generic-+` to extend the behavior of `binary-+` to any number of arguments, as shown in Figure 6.

The compiler macro is necessary, because we don't expect a compiler to be able to inline such a call to `reduce`. Without this compiler macro (or inlining of `reduce`, if it would succeed) there wouldn't be any compile-time type information at the call site of `binary-+` and, consequently, no optimization.

Now, with everything in place, we can use our new addition operator to define other functions. Figure 7 shows such a function, and the corresponding assembler code. We see that each call to the generic function `binary-+` can be reduced to a single assembler instruction. This observation is especially remarkable when we consider the size of the code that has been inlined, as shown in Figure 8.

But the real power of our generic addition operator is that we can still extend it outside of its sealed domain. This capability is illustrated in Figure 9.

³A real-world protocol would probably include additional generic functions for implicit coercion and dealing with commutativity.

```
5 (defun generic-+ (&rest things)
6   (cond ((null things) 0)
7         ((null (rest things)) (first things))
8         (t (reduce #'binary-+ things))))
9
10 (define-compiler-macro generic-+ (&rest things)
11   (cond ((null things) 0)
12         ((null (rest things)) (first things))
13         (t (reduce (lambda (a b) `(binary-+ ,a ,b))
14                     things))))
```

Figure 6: Extending `binary-+` to any number of arguments.

```
1 (defun add3 (x y z)
2   (declare (single-float x y z))
3   (generic-+ x y z))
```

```
mov RAX, [R13+16]
mov [RBP-8], RAX
movaps XMM1, XMM4
addss XMM1, XMM3 ; floating-point addition 1
addss XMM1, XMM2 ; floating-point addition 2
movd EDX, XMM1
shl RDX, 32
or DL, 25
mov RSP, RBP
clic
pop RBP
ret
```

Figure 7: A function using `generic-+` and the corresponding x86-64 assembler code on SBCL.

```
1 (lambda (#:x-7 #:y-8)
2   (let ((.gf. #'binary-+))
3     (macrolet ((call-method
4                 (method &optional next-methods)
5                 ...))
6       (flet ((next-method-p () nil)
7             (call-next-method ()
8              (no-next-method
               .gf.
               (specializer-prototype
                (find-class 'fast-method)))))
9         (funcall
10          (lambda (x y)
11            (block binary-+ (+ x y)))
12          #:x-7 #:y-8))))
```

Figure 8: The (slightly simplified) inline lambda generated for a single call to `generic-+` on two single floats.

```

1 (defmethod binary++ ((x string) (y string))
2   (concatenate 'string x y))
3
4 (generic++ "foo" "bar" "baz")
5 ;;; => "foobarbaz"

```

Figure 9: Extending binary++ to strings.

6.2 Extensible Sequence Functions

In this example, we will show how generic functions can be used to define an extensible sequence protocol. Our approach differs from the extensible sequence protocol by Rhodes [8] in that the sequence functions themselves are generic, with the exception of methods inside of their sealed domains.

Our exemplary sequence protocol consists of just three functions – generic-length, generic-elt and generic-count. The full source code of these functions is shown in Figure 10. Each generic function is extensible except for the domain of vectors. A more realistic protocol would provide more functions, additional keyword arguments, and an additional sealed domain for lists, but none of this is relevant for our explanation. As soon as the generic functions are defined, we use the trick from Figure 3 to generate specialized methods for all subclasses of vector.

The main point we want to make with this example is that fast generic functions can be nested. The definition of generic-count consists of calls to generic-length and generic-elt. Nevertheless, all this indirection disappears within a sealed domain, a fact that can be seen by looking at the assembler code for an exemplary function call as in Figure 11.

7 BENCHMARKS

Even though the assembler code in Figure 7 and Figure 11 looks promising in terms of performance, we have also conducted some further real-world benchmarks. We decided to compare the performance of the sequence function `cl:find` on SBCL with an equivalent definition that uses fast generic functions that we wrote for SICL [10], based on some earlier work by Durand and Strandh [5]. For our version, we present timings both with effective method inlining enabled and without. Furthermore, show timings for various element types and numbers of supplied keywords. These keywords have been chosen so that they supply the same values as the corresponding default values, i.e., we supply `#'eql` as the test function, `#'identity` as the key function, the sequence's length as the value of `:end` and `false` as the value of `:from-end`. Our benchmark results are shown in Figure 12.

The benchmark results are very promising. In particular, they show that eliminating the method dispatch and inlining the keyword argument parsing step can yield substantial benefits when working with short sequences. In almost all cases, our implementation of `find` is on par or even slightly faster than that of SBCL. The only cases where SBCL's implementation has a small lead is that of processing lists, and the case of having zero static information at the call site. We hope that we can speed up these cases in the future, too.

```

1 (defgeneric generic-length (sequence)
2   (:generic-function-class fast-generic-function))
3
4 (defgeneric generic-elt (sequence index)
5   (:generic-function-class fast-generic-function))
6
7 (defgeneric generic-count (item sequence &key test)
8   (:generic-function-class fast-generic-function))
9
10 (replicate-for-each-vectoroid #1#:#:vector
11   (defmethod generic-length
12     ((vector #1#))
13     (declare (method-properties inlineable))
14     (length vector)))
15
16 (defmethod generic-elt
17   ((vector #1#) (index integer))
18   (declare (method-properties inlineable))
19   (elt vector index))
20
21 (defmethod generic-count
22   (item (vector #1#) &key (test #'eql))
23   (declare (method-properties inlineable))
24   (loop for index below (generic-length vector)
25     for elt = (generic-elt vector index)
26     count
27     (funcall test item elt))))
28
29 (seal-domain #'generic-length '(vector))
30 (seal-domain #'generic-elt '(vector integer))
31 (seal-domain #'generic-count '(t vector))

```

Figure 10: A simple protocol for working with sequences and the corresponding inlineable methods for the domain of vectors.

But the important message for library authors is that a straightforward implementation using fast generic functions is at least as good as any highly specialized hand written dispatch mechanism, yet offers almost all the flexibility and convenience of standard generic functions. This was a very important goal for us, because it means programmers are not forced to choose between maintainability and performance anymore — they can have both.

8 CONCLUSIONS AND FUTURE WORK

We have presented a library for metaobject sealing that requires little more than access to the Metaobject Protocol of CLOS. In a second step, we have demonstrated how these metaobjects can be used to design fast generic functions. Finally, we have provided examples how these fast generic functions solve an infamous problem of Common Lisp — defining generic functions that are extremely efficient for a set of primitive types, yet extensible in general. We are especially proud that fast generic functions support the full set of

```

1 (defun generic-count-user (vector)
2   (declare (simple-vector vector))
3   (declare (optimize (safety 0))) ; simplify asm
4   (generic-count 5 vector :test #'equal))

mov RAX, [R13+16]
mov [RBP-8], RAX
mov RDI, [RSI-7] ; (length vector)
xor EAX, EAX ; (setf index 0)
xor ECX, ECX ; (setf count 0)
jmp L2
nop
L0: mov RBX, [RSI+RAX*4+1]; (generic-elt vector index)
cmp RBX, 10 ; (equal elt 5)
jeq L3
L1: add RAX, 2 ; (incf index)
L2: cmp RAX, RDI ; (= index (length vector))
j1 L0
mov RDX, RCX
mov RSP, RBP
clc
pop RBP
ret
L3: add RCX, 2 ; (incf count)
jmp L1

```

Figure 11: A function using generic-count and the corresponding x86-64 assembler code on SBCL. Thanks to effective method inlining, the compiler was able to eliminate even the keyword parsing step and could turn the call to equal to a single cmp instruction.

features of standard generic functions, including arbitrary method combinations and calls to `call-next-method` with arguments.

One issue we still have to address is performance portability. So far, the full set of fast generic function optimizations is only available on SBCL. At the very minimum, we'd like to port these optimizations to CCL and ECL. We are also willing to support closed-source implementations, given that someone can tell us how to access the necessary mechanisms in the compiler.

The other issue we are still working on is that of inheritance from multiple sealable classes. We have to ensure that doing so does not merge two previously disjoint domains of some generic functions. One way around this issue would be to allow only single inheritance of sealable classes, but we are trying to find a less restrictive technique.

This work was born out of our work on extensible sequence functions for the new Common Lisp implementation SICL[10]. Our hope is that what is useful to us is also useful to others. Experience reports and feedback are most welcome!

9 ACKNOWLEDGMENTS

We want to thank everyone on the #sicl IRC channel for their valuable feedback - especially Jan Moringen and Robert Strandh. Furthermore, we want to thank Pascal Costanza for writing and maintaining the invaluable CLOSER-MOP library.

element type	k	1 Element			50 Elements		
		SBCL	SICL	Inline	SBCL	SICL	Inline
*	0	30	32	32	447	342	343
*	1	36	60	60	454	371	372
*	2	39	87	87	454	397	396
*	4	51	140	140	466	507	490
single-float	0	20	17	2	422	360	181
single-float	1	20	18	6	444	354	213
single-float	2	21	18	9	445	354	305
single-float	4	21	21	9	436	406	474
list	0	15	15	5	404	424	175
list	1	17	17	7	422	422	263
list	2	17	21	9	402	585	224
list	4	18	23	9	574	696	337

Figure 12: Benchmark results comparing SBCL's built-in implementation of find with our implementation in SICL. All timings are given in nanoseconds. The first column describes the statically known vector element type, the second column describes the number of supplied keyword arguments, and the following two groups of three columns describe the results for processing a single element and 50 elements, respectively. Therein, the SBCL columns show the results on SBCL 2.0.1, the SICL columns show the results of our implementation that uses fast generic functions and no inlining, and the Inline columns show the results of our implementation but with inlining enabled.

REFERENCES

- [1] Masataro Asai. The inlined-generic-functions library. <https://github.com/guicho271828/inlined-generic-function>, 2015.
- [2] Henry G. Baker. Clostrophobia: Its etiology and treatment. *SIGPLAN OOPS Mess.*, 2(4):4–15, October 1991. ISSN 1055-6400. doi: 10.1145/126983.126984.
- [3] Pascal Costanza. The closer-mop library. <https://github.com/pcostanza/closer-mop>, 2005.
- [4] Mark Cox. The specialization-store library. <https://github.com/markcox80/specialization-store>, 2015.
- [5] Irène Durand and Robert Strandh. Fast, maintainable, and portable sequence functions. In *Proceedings of the 10th European Lisp Symposium on European Lisp Symposium*, ELS2017. European Lisp Scientific Activities Association, 2017.
- [6] Alexander Gutev. The static-dispatch library. <https://github.com/alex-gutev/static-dispatch>, 2018.
- [7] Gregor Kiczales. *The art of the metaobject protocol*. MIT Press, Cambridge, Mass, 1991. ISBN 978-0262610742.
- [8] Christophe Rhodes. User-extensible sequences in Common Cisp. In *Proceedings of the 2007 International Lisp Conference, ILC '07*, pages 13:1–13:14, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622138.
- [9] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison-Wesley Developers Press, Reading, Mass, 1996. ISBN 978-0201442113.
- [10] Robert Strandh. SICL: Building blocks for implementers of Common Lisp. In *Proceedings of the 4th European Lisp Symposium on European Lisp Symposium*, ELS2011. European Lisp Scientific Activities Association, 2011.