Sealable Metaobjects for Common Lisp

Marco Heisig

Motivation

```
(defgeneric two-arg-+ (a b)
  (:generic-function-class fast-generic-function)
  (:method two-arg-+ ((a float) (b float)
    (declare (method-properties inlineable))
    (+ a b))
  (:method two-arg-+ ((a number) (b number) …))
  (:method two-arg-+ ((a string) (b string) …))

(seal-domain #'two-arg-+ '(number number))
```

- Inline expansion for arguments that are floats.
- Fast calls for arguments that are numbers.
- Regular generic function call otherwise.

Project History

28.10.2018

beach: I figured out a few things that interested people could help me with, if they want to, like astalla or heisig. One thing would be to finish the implementation of the sequence functions, […]

29.01.2019

Idea to implement sequence functions via suitably restricted generic functions. The yak shave begins!

27.04.2020

- Quicklisp library #1: sealable-metaobjects
- Quicklisp library #2: fast-generic-functions
- Sequence functions are not finished.

Introduction

The AMOP

- Published in 1991
- *de facto* standard for CLOS

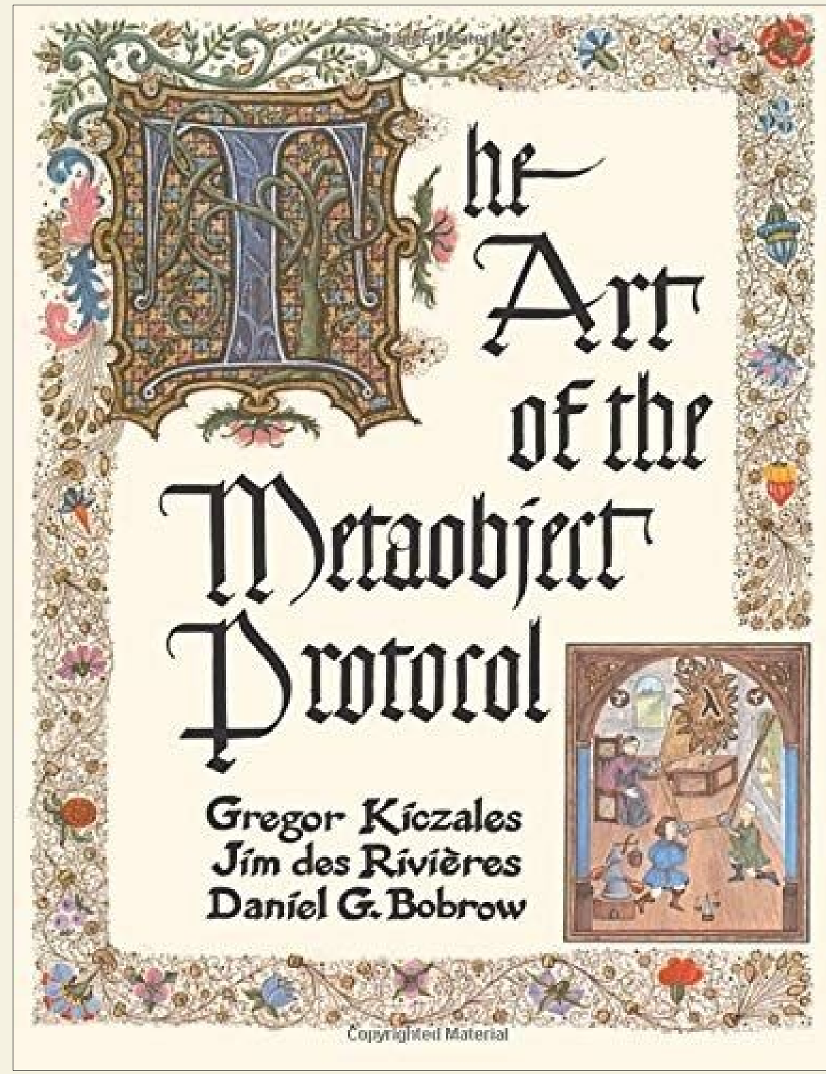
Additional Resources

- Closer MOP

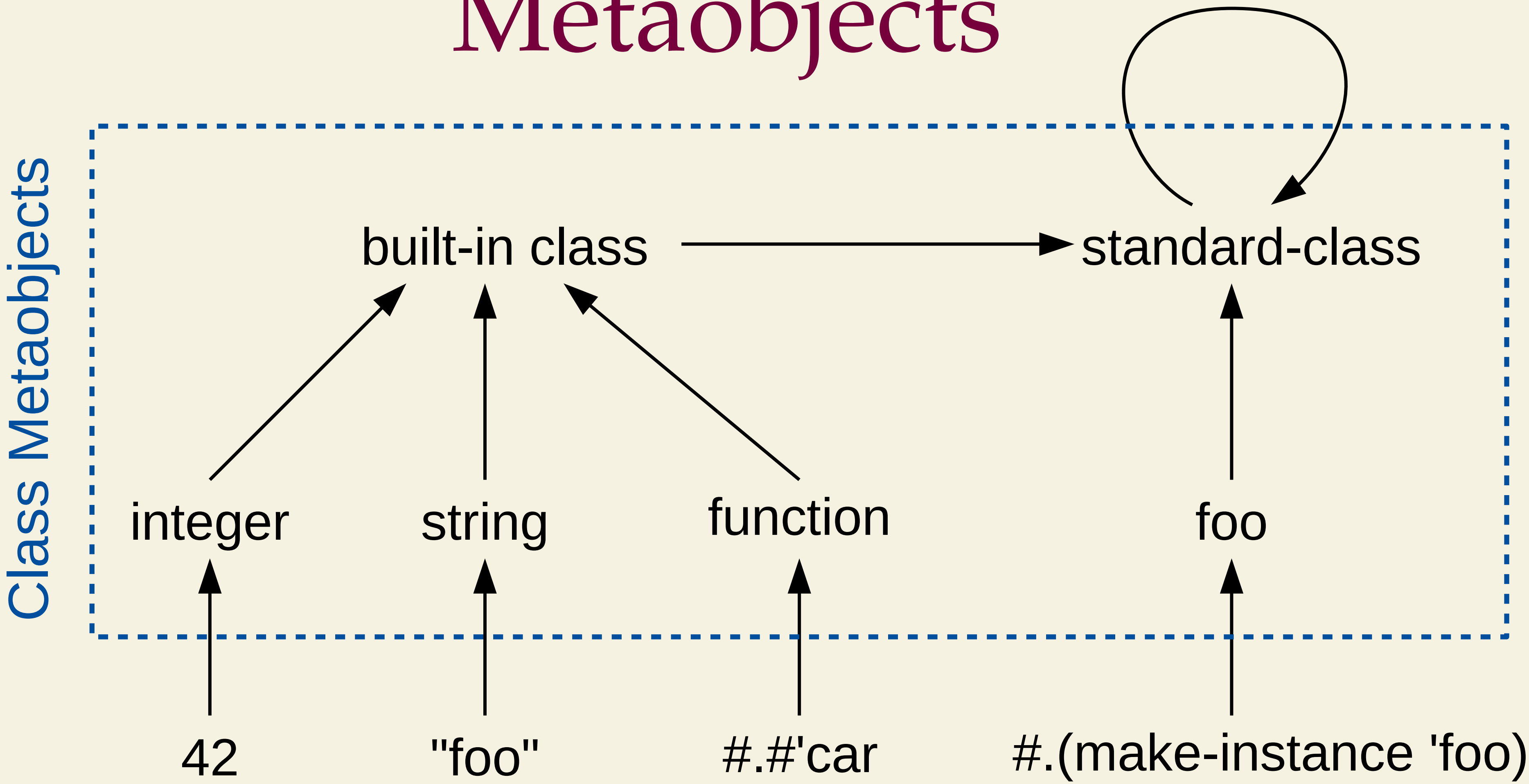
```
(ql:quickload :closer-mop)
```

- HTML Reference:

http://metamodular.com/CLOS-MOP



Metaobjects



Notation: A → B "A is an instance of B"

The AMOP defines generic function, method, slot-definition, method-combination, class, and eql-specializer metaobjects.

A Generic Function Call

1. Call to the discriminating function.
2. Computation of all applicable methods.
3. Computation of the effective method.
4. Invocation of the correct effective method.

- Typically, steps 2. and 3. are cached.
- This cache is cleared when metaobjects are modified.
- We want to perform 2. and 3. statically, and we want to replace step 1. with step 4. when appropriate.
- We can only do so (safely) if all involved metaobjects are sealed.

```
(defclass sealable-metaobject-mixin ()
  ((%sealed-p
    :initform nil
    :reader metaobject-sealed-p)))
```

Metaobject Sealing

```
(defclass sealable-generic-function
     (sealable-metaobject-mixin generic-function)
  ((%sealed-domains
    :initform '()
    :type list
    :accessor sealed-domains))
  (:default-initargs
   :method-class
   (find-class 'potentially-sealable-method))
  (:metaclass funcallable-standard-class))
```

Properties of Sealable Metaobjects

- A sealable metaobject has two states – sealed and unsealed.
- Once a sealable metaobject is sealed, it remains sealed.
- Calling reinitialize-instance on a sealed metaobject has no effect.
- It is an error to change the class of a sealed metaobject.
- It is an error to change the class of any object to a sealed metaobject.
- It is an error to change the class of an instance of a sealed metaobject.
- Each superclass of a sealed metaobject must be a sealed metaobject.

Note: System classes and structure classes fulfill these criteria.

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.

Example domain designators:

- `'(integer)`
- `'(string (eql 5))`
- `'(#<built-in-class single-float> #<eql-specializer 5.0>)`

Sealable Generic Functions

- A sealed generic function can have any number of sealed domains.
- New sealed domains can be added by calling `seal-domain`.
- All sealed domains of a generic function must be disjoint.
- Each method of a generic function must either be fully inside a sealed domain, or fully outside.
- Each method inside of a sealed domain must be sealed, and all its specializers must be sealed.
- It is an error to add or remove methods inside of a sealed domain.
- 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 (!).

Automatic Sealing

- When a sealable metaobject is sealed, all its superclasses are sealed automatically.
- When a sealable method is sealed, all its specializers are sealed automatically.
- The function `seal-domain` automatically seals the supplied generic function, and all methods inside of the designated domain.

Result:

- The distinction between sealed and unsealed metaobjects is mostly irrelevant to the user.
- Everything "just works".

Summary So Far

We have presented a library called *sealable-metaobjects* with the following properties:

- It provides the infrastructure for reasoning statically about both built-in, and user-defined objects and metaobjects.
- It defines the classes sealable-class, sealable-generic-function, and potentially-sealable-method.
- It provides the machinery for reasoning about generic function domains.
- It is fully portable and has a single dependency – closer-mop.

The second half of the talk is about how we can use these features to define fast generic functions.

```
(defclass fast-method
    (potentially-sealable-standard-method)
  (…))
```

Fast Generic Functions

```
(defclass fast-generic-function
    (sealable-standard-generic-function)
  (…)
  (:default-initargs
   :method-class (find-class 'fast-method))
  (:metaclass funcallable-standard-class))
```

Three Challenges

We face three challenges when statically optimizing certain calls to fast generic functions:

- Telling the compiler if and how to optimize a call to a sealed generic function.
- Computing the set of methods applicable to those types at compile time or at load time.
- Computing either an inlineable effective method, or a directly callable effective method function.

Bonus challenge:

- 100% portable code.

Compile Time Optimization #1

```
(defun fast-generic-function-compiler-macro (fgf)
  (lambda (form env)
    (block compiler-macro
      (dolist (s-d (sealed-domains fgf))
        (dolist (scs (compute-static-call-signatures fgf s-d))
          (when (loop for argument in (rest form)
                      for type in (static-call-signature-types scs)
                      always (compiler-typep argument type env))
            (return-from compiler-macro
              `(funcall ,(optimize-function-call fgf scs)
                        ,@(rest form))))))
      form)))

(defun compiler-typep (form type env)
  (or (constantp
       `(unless (typep ,form ',type)
          (tagbody label (go label)))
       env)
      (and (constantp form)
           (typep (eval form) type env))))
```

Compile Time Optimization #2

Unfortunately, our portable function for hooking into the compiler has some flaws:

- Slow – three nested loops over constantp and typep.
- Only works reliably for literal constants.
- Depends on compiler macros, which a compiler might ignore, especially for generic functions.

Instead, in practice, we use whatever mechanism an implementation provides, e.g., deftransform on SBCL.

Computing Applicable Methods

- We use the only sane way of computing all applicable methods, by calling `compute-applicable-methods`.
- The challenge is that `compute-applicable-methods` doesn't accept types or specializers, but arguments.
- Our solution is that we introduce *static call signatures*. A static call signature consists of a domain, a list of types, and a list of prototypes, each of the same length. The types denote a subset of the domain with a fixed set of applicable methods. Each prototype is of its corresponding type. The prototypes are chosen such that they unambiguously identify that particular subset of the domain.
- Choosing suitable prototypes is a challenge!

Computing the Effective Method

Good news:

There is a function called `compute-effective-method`

Bad news:

The result is a form containing "magic macros".

Possible Solution:

```
(defmethod f :around ((arg-1 t) …)
  (if *flag* #'call-next-method (call-next-method))
```

Actual Solution:

We expand the effective method ourselves, using our own versions of `call-method` and `make-method`.

Optimizations

We currently perform the following optimizations:

- Inlining of effective methods.
- Calling the effective method directly.
- Inlining of keyword parsing only.

Further optimizations are planned.

Examples & Benchmarks

SICL Sequence

```
(defclass sequence-function (fast-generic-function)
  ()
  (:metaclass funcallable-standard-class))

(defgeneric elt
    (sequence index)
  (:generic-function-class sequence-function))

(defgeneric length
    (sequence)
  (:generic-function-class sequence-function))

(defgeneric find
    (item sequence &key from-end test test-not start end key)
  (:generic-function-class sequence-function))

...
```

Interested?
https://github.com/robert-strandh/SICL/tree/master/Code/Sequence

Generic Find – Methods

```
(replicate-for-each-relevant-vectoroid #1=#:vectoroid
  (defmethod find (item (vectoroid #1#)
                   &key from-end test test-not (start 0) end key)
    (with-test-function (test test test-not)
      (with-key-function (key key)
        (for-each-relevant-element
            (element index vectoroid start end from-end)
          (when (test item (key element))
            (return-from find element)))))))

(seal-domain #'find '(t vector))
```

Details: *"Fast, Maintainable, and Portable Sequence Functions"*
by Irène Durand and Robert Strandh

Generic Find – Benchmarks

element type	k	1 Element SBCL	SICL	Inline	50 Elements 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

All timings are given in nanoseconds. We used SBCL version 2.0.1

Conclusions

- The library *sealable-metaobjects* can be used as a foundation for any project that attempts static reasoning about objects or metaobjects.
- The library *fast-generic-functions* is a drop-in replacement for any generic function that is used in performance-critical code.
- Fast generic function almost always outperform handcrafted solutions.
- Feedback and experience reports are most welcome!

Thank you for listening!

Questions or Suggestions?
marco.heisig@fau.de
https://github.com/marcoheisig
heisig on `#lisp`, `#sicl`, or `#petalisp`