# Extensible specializers and the CLOS Metaobject Protocol

## Implementation and Use

Christophe Rhodes

Goldsmiths, University of London

Monday 30th July

# Outline

# Motivation

```
<[1]solvent> hello. is it possible to make a single method
             that accepts two different classes of argument
      <H4ns> [1]solvent: no
         ...
       <Xof> I think it is possible to write methods with OR
             specializers
       <Xof> With a certain amount of wizardry
       <Xof> What I'm working out is how much wizardry
```

- What's so special about classes?
- Can we allow expression of algorithms naturally and maintainability?
- Does the CLOS Metaobject Protocol actually allow this kind of expressiveness in a controlled and composable way?
- If not, why not?

# Motivation

```
<[1]solvent> hello. is it possible to make a single method
             that accepts two different classes of argument
      <H4ns> [1]solvent: no
         ...
       <Xof> I think it is possible to write methods with OR
             specializers
       <Xof> With a certain amount of wizardry
       <Xof> What I'm working out is how much wizardry
```

- What's so special about classes?
- Can we allow expression of algorithms naturally and maintainability?
- Does the CLOS Metaobject Protocol actually allow this kind of expressiveness in a controlled and composable way?
- If not, why not?

# Motivation

```
<[1]solvent> hello. is it possible to make a single method
             that accepts two different classes of argument
      <H4ns> [1]solvent: no
         ...
       <Xof> I think it is possible to write methods with OR
             specializers
       <Xof> With a certain amount of wizardry
       <Xof> What I'm working out is how much wizardry
```

- What's so special about classes?
- Can we allow expression of algorithms naturally and maintainability?
- Does the CLOS Metaobject Protocol actually allow this kind of expressiveness in a controlled and composable way?
- If not, why not?

# Metamotivation

Experiment:

- can we get Common Lispers to agree on anything?

Theme: generate and implement language extensions, with a minimum of backwards incompatibility, and see if and how they are used.

- see also: generic sequences

"any useful lisp program is doomed to be made portable."

# Metamotivation

Experiment:

- can we get Common Lispers to agree on anything?

Theme: generate and implement language extensions, with a minimum of backwards incompatibility, and see if and how they are used.

- see also: generic sequences

"any useful lisp program is doomed to be made portable."

# Metamotivation

Experiment:

- can we get Common Lispers to agree on anything?

Theme: generate and implement language extensions, with a minimum of backwards incompatibility, and see if and how they are used.

- see also: generic sequences

“any useful lisp program is doomed to be made portable.”

# CLOS

Common Lisp Object System: Standardized by ANSI.

(Historical footnote: Common Lisp was the first ANSI-standardized language with Object Oriented features.)

- Objects are instances of Classes
- Objects may have Slots
- Inheritance is mediated through Classes
- Generic Functions take Object arguments
- Methods implementing behaviour belong to Generic Functions
- Methods are applicable to Arguments
- Methods are combined to form the Effective Method
- ...
- Generic Functions, Classes and Methods are Objects (and so is everything else)

# CLOS

Common Lisp Object System: Standardized by ANSI.

(Historical footnote: Common Lisp was the first ANSI-standardized language with Object Oriented features.)

- Objects are instances of Classes
- Objects may have Slots
- Inheritance is mediated through Classes
- Generic Functions take Object arguments
- Methods implementing behaviour belong to Generic Functions
- Methods are applicable to Arguments
- Methods are combined to form the Effective Method
- ...
- Generic Functions, Classes and Methods are Objects (and so is everything else)

# CLOS

Common Lisp Object System: Standardized by ANSI.

(Historical footnote: Common Lisp was the first ANSI-standardized language with Object Oriented features.)

- Objects are instances of Classes
- Objects may have Slots
- Inheritance is mediated through Classes
- Generic Functions take Object arguments
- Methods implementing behaviour belong to Generic Functions
- Methods are applicable to Arguments
- Methods are combined to form the Effective Method
- ...
- Generic Functions, Classes and Methods are Objects (and so is everything else)

# CLOS: Metaobject Protocol

CLOS:

- implemented with metacircles;
- base CLOS standardized by ANSI / X3J13; Metaobject Protocol (MOP) not recommended for standardization.
- we have a book instead: *The Art of the Metaobject Protocol* (AMOP).

MOP:

- introspection: `generic-function-methods`, `method-qualifiers`
- intercession: specifying
  - which functions are extensible or overrideable
  - when functions are called at particular stages in class realization, generic function call, etc.

# CLOS: Metaobject Protocol

CLOS:

- implemented with metacircles;
- base CLOS standardized by ANSI / X3J13; Metaobject Protocol (MOP) not recommended for standardization.
- we have a book instead: *The Art of the Metaobject Protocol* (AMOP).

MOP:

- introspection: `generic-function-methods`, `method-qualifiers`
- intercession: specifying
  - which functions are extensible or overrideable
  - when functions are called at particular stages in class realization, generic function call, etc.

# CLOS: Methods

```
(defmethod foo :after ((x integer) (y (eql 'foo)))
  (format *trace-output* "~&~S: ~X~%" x y))
```

```
#<STANDARD-METHOD
   :GENERIC-FUNCTION #<STANDARD-GENERIC-FUNCTION FOO>
   :QUALIFIERS (:AFTER)
   :SPECIALIZERS (#<STANDARD-CLASS INTEGER>
                  #<EQL-SPECIALIZER-OBJECT FOO>)
   :FUNCTION #<FUNCTION (LAMBDA (ARGS NEXT-METHODS))>
   ...>
```

# CLOS: Methods

```
(defmethod foo :after ((x integer) (y (eql 'foo)))
  (format *trace-output* "~&~S: ~X~%" x y))

#<STANDARD-METHOD
   :GENERIC-FUNCTION #<STANDARD-GENERIC-FUNCTION FOO>
   :QUALIFIERS (:AFTER)
   :SPECIALIZERS (#<STANDARD-CLASS INTEGER>
                  #<EQL-SPECIALIZER-OBJECT FOO>)
   :FUNCTION #<FUNCTION (LAMBDA (ARGS NEXT-METHODS))>
   ...>
```

# CLOS: Generic Functions

Generic functions have a set of methods and a method combination.
When a generic function is called:

1. All applicable methods are selected;
2. Applicable methods are sorted by precedence;
3. Method combination is applied to the sorted applicable methods.

CLHS 7.6.6.1

# Outline

# Desiderata

User-generalizeable specializers

- Usefulness
- Convenience
- Minimize incompatibility with existing standards
- Implementability

# Example

```
(defgeneric simplify (x)
  (:method (x) x))
(defmethod simplify ((x (+ _ 0)))
  (simplify (second x)))

(simplify '(+ (+ 1 0) 0)) ; => 1

(defmethod simplify ((x (* _x 0)))
  _x)
```

# Example

```
(defgeneric simplify (x)
  (:method (x) x))
(defmethod simplify ((x (+ _ 0)))
  (simplify (second x)))

(simplify '(+ (+ 1 0) 0)) ; => 1

(defmethod simplify ((x (* _x 0)))
  _x)
```

# MOP features

1. All applicable methods are selected;
2. Applicable methods are sorted by precedence;
3. Method combination is applied to the sorted applicable methods.

1. Discriminating function (returned by `compute-discriminating-function`);
2. `compute-applicable-methods` (and `compute-applicable-methods-using-classes`);
3. `compute-effective-method`.

Additionally MOP defines a `specializer` class.

# Example: Class definition

Subclass `specializer` class:

```
(defclass pattern-specializer (mop:specializer)
  ((pattern :initarg pattern :reader pattern)
   (%dms :initform nil
         :reader mop:specializer-direct-methods)))
```

Now what? Make a method!

```
(let ((s (make-instance 'pattern-specializer
                        :pattern '(+ _ 0))))
  (eval `(defmethod simplify ((x ,s))
           (simplify (second x)))))
```

# Example: Class definition

Subclass `specializer` class:

```
(defclass pattern-specializer (mop:specializer)
  ((pattern :initarg pattern :reader pattern)
   (%dms :initform nil
         :reader mop:specializer-direct-methods)))
```

Now what? Make a method!

```
(let ((s (make-instance 'pattern-specializer
                        :pattern '(+ _ 0))))
  (eval `(defmethod simplify ((x ,s))
           (simplify (second x)))))
```

# Example: Method definition

```
(let ((s (make-instance 'pattern-specializer
                        :pattern '(+ _ 0))))
  (eval `(defmethod simplify ((x ,s))
           (simplify (second x)))))
```

Ugly `eval`. What's the alternative?

```
(let* ((s (make-instance 'pattern-specializer
                         :pattern '(+ _ 0)))
       (f (lambda (a nm) (simplify (cadar a))))
       (m (make-instance 'standard-method
                         :qualifiers nil
                         :specializers (list s)
                         :function f)))
  (add-method #'simplify m))
```

# Example: Method definition

```
(let ((s (make-instance 'pattern-specializer
                        :pattern '(+ _ 0))))
  (eval `(defmethod simplify ((x ,s))
           (simplify (second x)))))
```

Ugly `eval`. What's the alternative?

```
(let* ((s (make-instance 'pattern-specializer
                         :pattern '(+ _ 0)))
       (f (lambda (a nm) (simplify (cadar a))))
       (m (make-instance 'standard-method
                         :qualifiers nil
                         :specializers (list s)
                         :function f)))
  (add-method #'simplify m))
```

# Example: Generic function

```
(simplify '(+ 3 0)) ; => ERROR
```

Applicability? Ordering? Method combination?

```
(defclass pattern-gf (standard-generic-function)
  ())
(defmethod compute-discriminating-function
    ((gf pattern-gf))
  (let ((ms (generic-function-methods gf)))
    (interpret-methods ms)))
```

# Example: Generic function

```
(defmethod compute-discriminating-function
    ((gf pattern-gf))
  (let ((ms (generic-function-methods gf)))
    (interpret-methods ms)))

(defun interpret-methods (methods)
  (lambda (&rest args)
    (dolist (m methods)
      (when (matches m args)
        (funcall (method-function m)
                 args nil)))))
```

# Usefulness and Convenience

Possible to make and call generic functions with non-standard specializers. What about easy?
New operators:

- `make-method-specializers-form`
- `parse-specializer-using-class`
- `unparse-specializer-using-class`

Methods on these operators permit the system to behave as one might desire.

```
(defgeneric simplify (x)
  (:generic-function-class pattern-gf/1))
(defmethod simplify ((y _)) y)
(defmethod simplify ((x (* _ 0))) 0)
```

# Example: Issues

Skated over many complicated issues in generic function protocol:

- multiple arguments;
- precedence ordering;
- method combination.

Also not discussed issues for the specializer implementor:

- Implementation of one specializer for general use must define interaction with standard specializers
- (Implementation of multiple specializer classes intended to be composed with arbitrary others must define a protocol for ordering.)

# Incompatibility

No known incompatibility in this with ANSI CL or the Metaobject Protocol described in AMOP.

Compatibility with Lisp programmers remains to be seen.

# Incompatibility

No known incompatibility in this with ANSI CL or the Metaobject Protocol described in AMOP.

Compatibility with Lisp programmers remains to be seen.

# Implementability

This is all implementable! *Proof*: SBCL 1.0.7 (June 2007).
Other Common Lisp implementations:

- CMUCL, GCL: straightforward port (similar codebase).
- CLISP: no way of getting a non-standard specializer into a method.
- Allegro: can trick it, but basically unsupported.
- Lispworks: `mop:specializer` not present.
- OpenMCL: different generic function calling protocol
- ECL, Corman: do not (claim to) support MOP

# Implementability

This is all implementable! *Proof*: SBCL 1.0.7 (June 2007).
Other Common Lisp implementations:

- CMUCL, GCL: straightforward port (similar codebase).
- CLISP: no way of getting a non-standard specializer into a method.
- Allegro: can trick it, but basically unsupported.
- Lispworks: `mop:specializer` not present.
- OpenMCL: different generic function calling protocol
- ECL, Corman: do not (claim to) support MOP

# Outline

# Conclusions

Classes are a bit special:

- given an instance, natural ordering for classes.
- (slightly different from issue of CPL ordering in the first place)

Specializers can usefully be subclassed:

- express some algorithms more straightforwardly
- potentially more efficient than simple, static implementations.

No-one is developing with extended specializers (yet)

- You can be the first!

# Future Work

- Get specializers used (and implemented for other CL implementations)
- Feedback from use might suggest suitable protocols for interoperable specializers

# Summary

Resources:

- SBCL home page: `http://www.sbcl.org/`
- Manual: `http://www.sbcl.org/manual/`
- MOP: `http://www.lisp.org/mop/`

Aiming higher than a late-1980s programming language.