Custom Specializers in Object-Oriented Lisp

Jim Newton and Christophe Rhodes

Cadence Design Systems and Goldsmiths, University of London

23rd May 2008
Common Lisp:

- Unification of various Lisp dialects;
- Not dead yet.

Skill:

- Internal Lisp dialect from Cadence Design Systems;
- Extension language for Integrated Circuit software;
- Optional C-style syntax;
- Simple object system (Skill++, inspired by CLOS).
Object-oriented Lisp

CLOS and VCLOS

- CL structures and Skill++ objects:
  - single inheritance
  - single dispatch

- CL and VCLOS standard-objects and generic functions
  - multiple inheritance
  - multiple dispatch
  - method combination
  - [and lots more goodies]

Both CLOS and VCLOS are (almost) implementable as extensions to the base language

- In Common Lisp, CLOS is specified and fully-integrated;
- In Skill, VCLOS is an extension library.
Object-oriented Lisp
CLOS and VCLOS

- CL structures and Skill++ objects:
  - single inheritance
  - single dispatch
- CL and VCLOS standard-objects and generic functions
  - multiple inheritance
  - multiple dispatch
  - method combination
  - [and lots more goodies]

Both CLOS and VCLOS are (almost) implementable as extensions to the base language

- In Common Lisp, CLOS is specified and fully-integrated;
- In Skill, VCLOS is an extension library.
Object-oriented Lisp
CLOS and VCLOS

- CL structures and Skill++ objects:
  - single inheritance
  - single dispatch
- CL and VCLOS standard-objects and generic functions
  - multiple inheritance
  - multiple dispatch
  - method combination
  - [and lots more goodies]

Both CLOS and VCLOS are (almost) implementable as extensions to the base language

- In Common Lisp, CLOS is specified and fully-integrated;
- In Skill, VCLOS is an extension library.
VCLOS is implemented in itself; CLOS is ‘encouraged’ to be

- Metaobject Protocols
- Introspect and intercede...
- ...using the (V)CLOS mechanisms themselves.

Examples:

- customize slot access: integration with ‘foreign’ object systems;
- customize generic function call: integration with other languages;
- customize class precedence computation: run old codebases.
(defclass class (specializer)
  ((name :initarg :name :reader class-name)
   (direct-slots :initarg :direct-slots :reader class-direct-slots)
   ...))

(defclass eql-specializer (specializer)
  ((object :initarg :object :reader eql-specializer-object)))

What if we could do...

(defclass equal-specializer (specializer)
  ((object :initarg :object :reader equal-specializer-object)))

... and how hard is it to make this work?
Example

Subclassing the specializer metaobject class

(defclass class (specializer)
  ((name :initarg :name :reader class-name)
   (direct-slots :initarg :direct-slots :reader class-direct-slots)
   ...))

(defclass eql-specializer (specializer)
  ((object :initarg :object :reader eql-specializer-object)))

What if we could do...

(defclass equal-specializer (specializer)
  ((object :initarg :object :reader equal-specializer-object)))

... and how hard is it to make this work?
(defmethod walk ((expr list) env call-stack)  
  (let ((call-stack (cons expr call-stack)))  
    (walk (car expr) env call-stack)  
    (walk (cdr expr) env call-stack)))

(defmethod walk ((form (cons (eql 'quote))) env call-stack)  
  nil)

(defmethod walk ((form (cons (eql 'lambda))) env call-stack)  
  (destructuring-bind (lambda lambda-list &rest body) form  
    (let ((bs (derive-bindings-from-ll lambda-list)))  
      (dolist (form body)  
        (walk form (make-env bs env) (cons form call-stack)))  
      (dolist (bind bs)  
        (unless (used bind)  
          (format t "unused: ~A: ~A~%" var call-stack)))))
Example
Code walker dispatch

(defmethod walk ((expr list) env call-stack)
  (let ((call-stack (cons expr call-stack)))
    (walk (car expr) env call-stack)
    (walk (cdr expr) env call-stack)))

(defmethod walk ((form (cons (eql 'quote))) env call-stack)
  nil)

(defmethod walk ((form (cons (eql 'lambda))) env call-stack)
  (destructuring-bind (lambda lambda-list &rest body) form
    (let ((bs (derive-bindings-from-ll lambda-list)))
      (dolist (form body)
        (walk form (make-env bs env) (cons form call-stack)))
      (dolist (bind bs)
        (unless (used bind)
          (format t "unused: ~A: ~A~%" var call-stack)))))))
Example

Symbolic simplification

(defgeneric simplify (x)
  (:method (x) x))

;;; in plus.lisp
(defmethod simplify ((x (+ _ 0)))
  (simplify (second x)))
(defmethod simplify ((x (+ 0 _)))
  (simplify (third x)))

;;; in times.lisp
(defmethod simplify ((x (* _ 0)))
  0)
(defmethod simplify ((x (* _ 1)))
  (simplify (second x)))
...

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

XML whitespace normalization

;;; default method: do nothing
(defmethod process-node ((node t) (strip-space-p nil))
  (declare (ignore strip-space-p)))

;;; strip this text node if it contains whitespace only
(defmethod process-node ((node stp:text) (strip-space-p (eql t)))
  (when (whitespace-only-p (stp:data node))
    (stp:detach node)))

;;; process children recursively for document and element nodes
(defmethod process-node ((node stp:parent-node) (strip-space-p nil))
  (mapc (lambda (child)
            (process-node child strip-space-p))
        (stp:list-children node)))

;;; override the stripping mode when declared explicitly on a element:
(defmethod process-node ((node (xpattern "*@xml:space = 'preserve'"))
                        strip-space-p)
  (declare (ignore strip-space-p))
  (call-next-method node nil))

(defmethod process-node ((node (xpattern "*@xml:space = 'strip'"))
                        strip-space-p)
  (declare (ignore strip-space-p))
  (call-next-method node t))
Run-time generic functions:

- `parse-specializer-using-class gf name`
- `unparse-specializer-using-class gf spec`

for `find-method`, debugger, tracer

`defmethod-time` generic function

- `make-method-specializers-form gf method names env`

(minimum necessary: more fine-grained protocol needed for convenience)
Run-time generic functions:

- `parse-specializer-using-class gf name`
- `unparse-specializer-using-class gf spec`

for `find-method`, debugger, tracer

`defmethod-time` generic function

- `make-method-specializers-form gf method names env`

(minimum necessary: more fine-grained protocol needed for convenience)
Discriminating functions are the ‘function’ part of a generic function.

```
(defmethod compute-discriminating-function ((gf generic-function))
  (lambda (&rest args)
    (let* ((ams (compute-applicable-methods gf args))
           (mc (generic-function-method-combination gf))
           (em (compute-effective-method gf mc ams))
           (emf (generate-effective-method-function em)))
      (apply emf args))))
```

We need to override at least compute-applicable-methods...
Discriminating functions are the ‘function’ part of a generic function.

```lisp
(defmethod compute-discriminating-function ((gf generic-function))
  (lambda (&rest args)
    (let* ((ams (compute-applicable-methods gf args))
           (mc (generic-function-method-combination gf))
           (em (compute-effective-method gf mc ams))
           (emf (generate-effective-method-function em)))
      (apply emf args))))
```

We need to override at least `compute-applicable-methods`
(defmethod compute-applicable-methods ((gf generic-function) args)
  (sort
    (remove-if-not (applicable-predicate args)
      (generic-function-methods gf))
    (ordering-function gf args)))

Doing this on every generic function call would be slow.

Regular CLOS MOP has paired operators:

1. compute-applicable-methods
2. compute-applicable-methods-using-classes

and caching is done if compute-applicable-methods-using-classes can work out the answer.
(defmethod compute-applicable-methods ((gf generic-function) args)
  (sort
   (remove-if-not (applicable-predicate args)
     (generic-function-methods gf))
   (ordering-function gf args)))

Doing this on every generic function call would be slow.

Regular CLOS MOP has paired operators:

1. compute-applicable-methods
2. compute-applicable-methods-using-classes

and caching is done if c-a-m-using-classes can work out the answer.
(defmethod compute-discriminating-function ((gf generic-function))
  (lambda (&rest args)
    (let* ((ams (compute-applicable-methods gf args))
           (mc (generic-function-method-combination gf))
           (em (compute-effective-method gf mc ams))
           (emf (generate-effective-method-function em)))
      (apply emf args))))

- CLOS MOP specifies compute-effective-method
- compute-effective-method-function would be more useful
- ... and also protocol for automatically
  - cacheing the effective method;
  - clearing the cache;
  - pre-filling the cache...
Expressivity:

- clarity;
- modularity;
- dynamicity.

Efficiency:

```lisp
(defmethod compute-discriminating-function ((gf generic-function))
  (let* ((methods (generic-function-methods gf))
          (dfun (compile-dispatch-function methods)))
    (lambda (&rest args)
      (apply dfun args))))
```
Expressivity:

- clarity;
- modularity;
- dynamicity.

Efficiency:

```
(defmethod compute-discriminating-function ((gf generic-function))
  (let* ((methods (generic-function-methods gf))
          (dfun (compile-dispatch-function methods)))
    (lambda (&rest args)
      (apply dfun args))))
```
Conclusions

- Custom specializers are an additional tool in the CLOS toolchest;
- Skill/VCLOS experience shows that they can be useful;
- The basic machinery has been implemented in a Common Lisp;
- There are details in the protocol to be sorted out to make it easy to use.
Future Work

- Usable protocol for compute-applicable-methods
  - specializer-accepts-class-p
  - specializer<

- Better cacheing protocol
  - specializer-of
  - compute-applicable-methods-using-specializers
  - invalidation on redefinition of dependents

- Compelling applications
  - efficient ML-style pattern matcher with run-time dynamicity
  - dispatch based on emacs-like ‘active modes’
  - [your favourite here]
• Please download and try: available in current SBCLs!
• For the Skill version, contact your nearest Cadence Design Systems representative.

Thanks:
• Cadence Design Systems
• Goldsmiths College
• David Lichteblau
• Paul Khuong
• Pascal Costanza