Reflection, Reflective Interpreters and Meta-Object Protocols (MOP's)

6.894 (OODL) Spring 2001

Gregory T. Sullivan
MIT AI Lab

Reflection and Reflective Interpreters

Introspection: How, what am I doing? Structural, behavioral.

Reification – making aspects of the interpreter "real".

Intercession: Changing behavior.

Reflection – reflecting program logic into the interpreter.

How to give a program access to the internals of the interpreter currently interpreting the program?

- *Reflective Interpreters* as in [Smith 84].
- See *A Simple Reflective Interpreter* by Jefferson and Friedman, 1992.
- *Reifying Procedures* inject user code into the interpreter, gaining access to environment and continuation.

Examples in $\mathcal{I}_{\mathcal{R}}$

```
(set! when
  (compound-to-reifier
    (lambda (e r k)
       (evaluate (1st e) r
          (lambda (v)
             (if v
                 (evaluate (2nd e) r k)
                 (k (void))))))))
```

Similar for `call-with-current-continuation`.

Changing interpreter semantics in $\mathcal{I}_{\mathcal{R}}$ is all too easy, as global environment is shared and includes definitions of evaluate, etc.

MetaJ

By Rémi Douence and Mario Südholt, HOSC 2001

```
class Pair {
    String fst;
    String snd;
    Pair(String fst, String snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
class PrintablePair extends Pair {
    String toString() {
        return "(" + this.fst + "," +
          this.snd + ")";
    }
}
class InstanceWithTrace extends Instance {
    Method lookupMethod(String name) {
        // trace methodcalled
        System.out.println
          ("method called: " + name);
        return this.instanceLink.methodList
          ().lookup(name);
    }
}
```

```
class Main {
    void main() {
        Pair pair = new Pair("1", "2");
        // 1  invariance under reification
        Instance metaPair = delta(pair);
        System.out.println("pair.fst: " + pair.fst);
        // 2  introspection: test existence of a super class
        Class metaClass = delta(Pair);
        if (metaClass.getExtendsLink() == null)
            System.out.println("Class Pair has no superclass");
        // 3  intercession: dynamic class change
        metaPair.instanceLink = PrintablePair;
        System.out.println("pair.toString(): " + pair.toString());
        // 4  intercession: change methodcall semantics
        Instance metaMetaPair = delta(metaPair);
        metaMetaPair.setInstanceLink(InstanceWithTrace);
        System.out.println("pair.toString(): " + pair.toString());
        // 5  instance and class deification
        System.out.println(tri-Instance(metaPair)).fst);
        reify(PrintablePair).extendsLink = tri-Class(metaClass);
        System.out.println("pair.toString(): " + pair.toString());
    }
}
```

Problems With Reflective Interpreters

- Fragile – directly expose internal workings of interpreter.
- Horrible performance.

Meta-Object Protocols

From *The Art of the Metaobject Protocol* (Kiczales, des Rivieres, Bobrow, 1991):

> What reflection on its own doesn't provide, however, is flexibility, incrementality, or ease of use. This is where object-oriented techniques come into their own. These techniques work by (i) defining a set of object types and operations on them, which can be called a protocol; (ii) defining a default behavior, a single point in the region, in terms of the protocol – this is the role of the default classes and methods; and (iii) making it possible to effect incremental adjustments from the default behavior to other points in the region – this is the role of inheritance and specialization.

"Backstage" Code

- `standard-class`
- `defclass` macro
- `compute-class-precedence-list`
- `standard-generic-function`
- `standard-method`

Generic Function Call Implementation

```
(apply-generic-function gf args)
  (compute-applicable-methods-using-classes gf classes)
    (method-more-specific-p gf m1 m2 classes)
  (apply-methods gf args ms)
    (apply-method m args next-ms)
      (extra-function-bindings m)
```

Introspection API

standard-class: class-name, class-direct-superclasses, class-direct-slots, class-precedence-list, class-slots, class-direct-subclasses, class-direct-methods.

slots: slot-definition-name, slot-definition-initargs, slot-definition-initfunction, slot-definition-initform, slot-definition-readers, slot-definition-writers

generics: generic-function-name, generic-function-lambda-list, generic-function-methods,

methods: method-generic-function, method-lambda-list, method-qualifiers, method-specializers, method-body, method-environment

Uses of Introspection

- Program Analysis
- Debugging
- Optimization?
- Serialization
- Components (a la JavaBeans)

Intercession

Example:

```
(defclass counted-class (standard-class)
  ((counter :initform 0)))

(defmethod make-instance :after ((class counted-class) &key)
  (incf (slot-value class 'counter)))

(defclass counted-rectangle (rectangle)
   ()
  (:metaclass counted-class))
```

Counted class ex. is on pg. 72-73.

Example: Monitoring Slot Access

Default implementation of `slot-value-using-class`:

```
(defmethod slot-value-using-class ((class standard-class)
                                   instance slot-name)
  (let* ((location (slot-location (class-of instance)
                                  slot-name))
         (slots (std-instance-slots instance))
         (val (slot-contents slots location)))
    (if (eq secret-unbound-value val)
        (error "The slot ~S is unbound in the object ~S."
               slot-name instance)
        val)))
```

slot monitoring ex. is pg.97

Monitoring Slot Access, cont'd

```
(defclass monitored-class (standard-class) ())

(defmethod slot-value-using-class :before
        ((class monitored-class) instance slot-name)
  (note-operation instance slot-name 'slot-value))

(defmethod (setf slot-value-using-class) :before
        (new-value (class monitored-class)
         instance slot-name)
  (note-operation instance slot-name 'set-slot-value))

(defmethod slot-boundp-using-class :before
        ((class monitored-class) instance slot-name)
  (note-operation instance slot-name 'slot-boundp))

(defmethod slot-makeunbound-using-class :before
        ((class monitored-class) instance slot-name)
  (note-operation instance slot-name 'slot-makeunbound))
```

Example: Counting Generic Function Applications

Recall the generic function invocation protocol:

```
(apply-generic-function gf args)
  (compute-applicable-methods-using-classes gf classes)
    (method-more-specific-p gf m1 m2 classes)
  (apply-methods gf args ms)
    (apply-method m args next-ms)
      (extra-function-bindings m)
```

Counting Generic and Method Invocations

We can create a "counting generic" as follows:

```
(defclass counting-gf (standard-generic-function)
    ((call-count :initform 0 :accessor call-count)))

(defmethod apply-generic-function :before
    ((gf counting-gf) args)
  (incf (call-count gf)))

(defclass counting-method (standard-method)
    ((call-count :initform 0 :accessor call-count)))

(defmethod apply-method :before
    ((method counting-method) args next-methods)
  (incf (call-count method)))
```

Protocol Design and Performance

AMOP distinguishes between:

Functional protocols (e.g. `compute-class-precedence-list`)
Return something (a value).

Procedural protocols (e.g. `apply-generic-function`)
Do something (are called for effect).

Functional protocols are more amenable to optimization (e.g. memoization)

⇒ Make as much of protocol functional as possible.

Effective Method Functions

Replace `apply-methods` procedure with `compute-effective-method-function`.

Implementation of `(apply-methods gf args methods)` becomes

```
(funcall (compute-effective-method-function gf methods)
         args)
```

Discriminating Functions

Same idea for `(apply-generic-function gf args)` is implemented as:

```
(apply (compute-discriminating-function gf) args)
```

The default implementation is produced directly from original definition of `apply-generic`, using currying:

```
(defmethod compute-discriminating-function
      ((gf standard-generic-function))
  #'(lambda (&rest args)
       (let ((applicable-methods
                (compute-applicable-methods-using-classes
                   gf
                   (mapcar #'class-of ...))))
         (if (null applicable-methods)
             (error ...)
             (apply-methods gf args applicable-methods)))))
```

Optimizing Discriminating Function

```
(defmethod compute-discriminating-function
      ((gf standard-generic-function))
  (let ((table (make-hash-table :test #'equal)))
     #'(lambda (&rest args)
         (let* ((classes (mapcar #'class-of
                                 (required-portion gf args)))
                (emfun (gethash classes table nil)))
          (if emfun
              (funcall emfun args)
              (slow-method-lookup gf args
                     classes table))))))

(defun slow-method-lookup (gf args classes table)
  (let* ((applicable-methods
           (compute-applicable-methods-using-classes gf classes))
         (emfun
           (compute-effective-method-function
               gf applicable-methods)))
    (setf (gethash classes table) emfun)
    (funcall emfun args)))
```

see pg. 130.

Discussion - Optimization of Original Protocol?

```
(apply-generic-function gf args)
  (compute-applicable-methods-using-classes gf classes)
    (method-more-specific-p gf m1 m2 classes)
  (apply-methods gf args ms)
    (apply-method m args next-ms)
      (extra-function-bindings m)
```

- Specialization
- Inlining
- CLOS MOP Dependent Maintenance Protocol