Reflection in Direct Style

Kenichi Asai

Ochanomizu University

October 23, 2011
Reflection: quest for language internals

Self-modifying code:
- peeks into machine code representation

Meta-object protocol (MOP)
- representation of objects open to user programs
- any modification to objects possible
- hard to implement efficiently

Aspect-oriented programming (AOP)
- handles many interesting uses of reflection
- restricted to supported aspects

Java reflection
- inspection of object representation only
- no modification
Theory of behavioral reflection

3-LISP [Smith84]
   Tower of meta-circular interpreters.
   Provides the general model of reflection.

Brown [FW84,WF86]

Blond [DM88]
   Meta-continuation.
   Implementable. Efficient.

Black [AMY96,A03]
   Interpreted code vs. compiled code.
   Hook to keep interpreters modifiable.
Theory of behavioral reflection

3-LISP [Smith84]
   Tower of meta-circular interpreters.
   Provides the general model of reflection.

Brown [FW84,WF86]
Blond [DM88]
   Meta-continuation.
   Implementable. Efficient.

Black [AMY96,A03]
   Interpreted code vs. compiled code.
   Hook to keep interpreters modifiable.
   Level-shifting anomaly.
3-LISP: tower of meta-circular interpreters

(define (eval exp env)
  (cond ((var? exp) (eval-var exp env))
        ((if? exp) (eval-if exp env))
        (...))

(define (eval-var exp env) ...)
(define (eval-if exp env) ...)
(define (eval-ap op args) ...)

Reflective tower: \( \rho \): an initial environment

1: (eval '⟨program⟩ \( \rho \))
2: (eval '(eval '⟨program⟩ \( \rho \)) \( \rho \))
3: (eval '(eval '(eval '⟨program⟩ \( \rho \)) \( \rho \)) \( \rho \))
Brown/Blond: meta-continuation

If `eval` is not modified, `(eval '(eval ... ρ) ρ)` is the same as `(eval ... ρ)`.

1: `(eval '<program> ρ)
2: `(eval '(eval '<program> ρ) ρ)
3: `(eval '(eval '(eval '<program> ρ) ρ) ρ)

Meta-continuation keeps state (env, cont) of each level.

- Implementable
- Efficient
- Interpreter is monolithic (compiled) and unmodifiable
The reflective language Black

Replaces (eval '(eval ... ρ) ρ') with (eval ... ρ) with a hook: meta-apply.

(define (eval exp env)
  (cond
   ((var? exp) (meta-apply 'eval-var exp env))
   ((if? exp) (meta-apply 'eval-if exp env))
   ...))

meta-apply consults the metalevel environment ρ' to see if the called function is modified.
meta-apply

(define (meta-apply name exp env)
  (shift k (shift-up (lambda (mr mk)
         (let ((op (cdr (get name mr))))
          (if (procedure? op)
            (shift-down (k (op exp env)) mr mk)
            (let ((x (eval-ap op (list exp env))))
               (shift-down (k x) mr mk)))))))

(eval 'x ρ)
-> (meta-apply 'eval-var 'x ρ)
-> (eval-var 'x ρ)
meta-apply

(define (meta-apply name exp env)
  (shift k (shift-up (lambda (mr mk)
      (let ((op (cdr (get name mr)))
        (if (procedure? op)
          (shift-down (k (op exp env)) mr mk)
          (let ((x (eval-ap op (list exp env))))
            (shift-down (k x) mr mk)))))))

(eval 'x \rho)
-> (meta-apply 'eval-var 'x \rho)
-> (let ((x (eval-ap op (list 'x \rho))))
    (shift-down (k x) \rho \rho \mk))
Launching a new level

(define (eval-ap op args env)
  (cond ((primitive? op) (apply op args))
    ((closure? op)
     interpret body of op)
    ((eval-fun? op)
     (shift k
      (shift-down (go-up (apply op args))
       env k))))

(eval '(eval 'x ρ) ρ′)
-> (eval-ap eval (list ’x ρ) ρ′)
-> (shift-down (go-up (eval ’x ρ)) ρ′ k)
Characteristics of Black

- Infinite reflective tower.
- Interpreters are modifiable.
- Default interpreters are efficiently executed.

Key techniques

- Distinction between compiled and interpreted code
- Use of hooks to make interpreters modifiable

**go up** when compiled code calls interpreted code.

**go down** when interpreted code calls compiled code.

Black is an intuitive and fun-to-use reflective language...
..., if no abortion occurs.

Abortion: exit or an error.

**Level-shifting anomaly**

Once the metalevel main function eval is modified, it becomes impossible to exit the current level.

```
(define old-eval eval)
(set! eval
  (lambda (exp env)
    (write exp) (newline)
    (old-eval exp env)))
```
Reason: lack of tail reflection optimization

(rep-loop 'a)
-> (meta-apply 'eval 'a ρ)
-> (let (((x (eval-ap op (list 'a ρ))))
  (shift-down (k x) ρ mk))
  (prints a)
-> (let (((x (eval-ap old-eval (list 'a ρ))))
  (shift-down (k x) ρ mk))
-> (let (((x (go-up (old-eval 'a ρ))))
  (shift-down (k x) ρ mk))
Technical contribution of the paper

- Showed that the lack of tail reflection optimization leads to level-shifting anomaly.
- Implemented Black without level-shifting anomaly by
  - transforming the internal default functions into CPS,
  - defunctionalizing continuations, and
  - cancelling out going-up and -down.

  cf. The user-observable metalevel interpreter is still in direct style.

The first fully reflective language that is usable

  cf. The previous CPS Black required that the metalevel interpreter be written in CPS.
Perspective

For efficient execution:
- Compilation via partial evaluation.
  \[(\text{eval } (\text{eval } (\text{eval } \langle \text{program} \rangle \ \rho) \ \rho) \ \rho)\]

Meta-circularity of user-observable interpreter:
- shift/reset are used in the interpreter.
- They are not supported by the interpreter.
- Supporting them appears to require shift2.

Reflective tower vs. CPS hierarchy?
- Relationship still not clear.