REFLECTION

Reflection

Meta programming: reasoning about a computational system
Reflection: the same language used to reason and act over itself

- Terms related to self-reference:
  - Reflection
  - Meta-level
  - Multi-level
  - Reification
  - Meta-logic, etc.

General Theory of reference seeks to study, with universal theoretic inclusiveness, an essential constitutive ingredient of human reality: the phenomenon of referring which, in different forms, is involved in all study, all reflection, all discourse. It appears to be an inescapable fundamental basis of all that can be thought and expressed.

(Barlett i Suber, 1987)

Self-reference

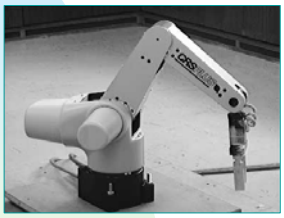
- BIOLOGY
  - Self-organization
- LINGUISTICS
  - Natural language
- MUSIC
  - Cyclic structures, fugue, canon.
- ART
  - Painting into a painting
- Psychiatry
  - Reflection capacity of patient
- GENERAL SYSTEMS THEORY
  - feedback.
- PROOF THEORY
  - Incompleteness, decidability

- ARTIFICIAL INTELLIGENCE
  - Self-awareness
  - Self-configuration
  - Self-diagnosis
  - Self-correction
  - Learning
  - Self-organization
  - Self-reproduction
  - Self-repair
- Others
  - General programming: data/program
  - Logic programming: theory/metatheory
  - Partial evaluation

Reflection

Causal connection: internal structures and the domain (represented by internal structures) are linked: is one of them changes there is a consistent corresponding effect on the other.

DOMAIN



REPRESENTATION

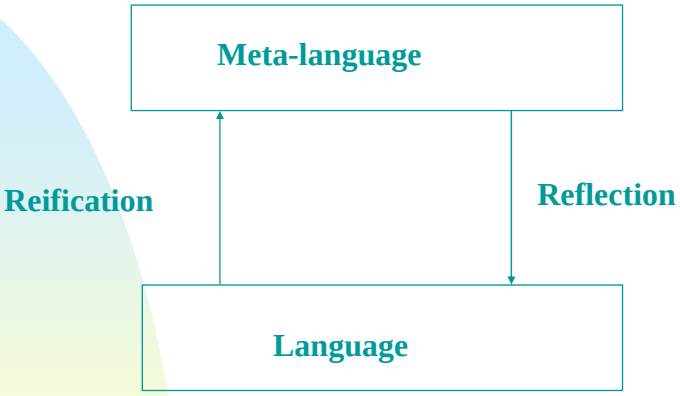
Causal connection

(Coordinates x y)
(Height z)

REFLECTIVE SYSTEM: contains structures representing aspects of itself

- A reflective system incorporates structures representing itself (self-representation). It is possible to answer questions about itself and support actions on itself.
- The representation is causally connected to the system:
  1. The system always has an accurate representation of itself.
  2. The computation of the system is always in compliance with its representation.

General scheme of reflection



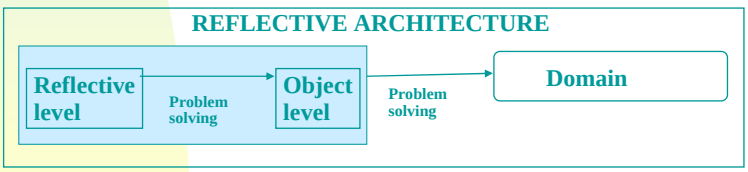
(Language = Meta-language) → Reified Language

Reflective facilities in languages

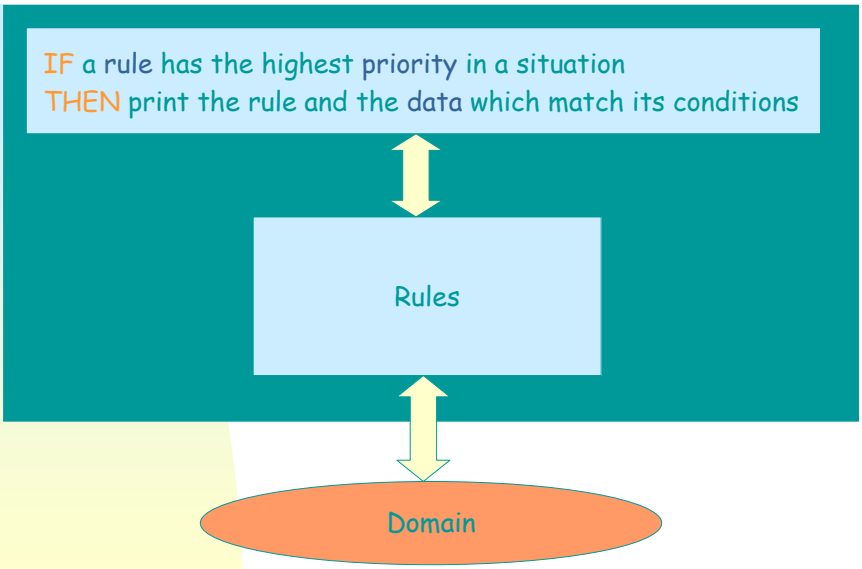
- No facilities (ex: debugging with extra statements)
- Fixed facilities (Lisp)
  - Programs as data: eval, apply
  - Run-time stack: catch & throw
  - Run-time environment: boundp
  - Interpreter
- Reflective systems
  - Explicit
    - 3-Lisp, BROWN
    - FOL, Meta-Prolog
    - 3-KRS
  - Implicit
    - TEIRESIAS, SOAR, MILORD

Reflective Architecture

- A programming language with reflective architecture (meta-level) uses reflection as a fundamental programming concept
- Distinction between *classical* and AI languages.
- Characteristics of a reflective architecture:
  1. The interpreter has to give access to data representing (aspects of) the system itself.
  2. Programmers write code to manipulate these data.
  3. The interpreter has to guarantee the causal connection between these data and the system they represent.
- New paradigm



Example: trace



Procedural-based example

- Reflective functions
  - Arguments
  - Environment (list of bindings)
  - Continuation

```
(define-reflect boundp-else-bind-to-one (symbol &optional env cont)
    (let ((value (binding symbol env)))
      (funcall cont
               (if value
                   value
                   (rebind symbol 1 env)))))

(let ((x 36) (y 12) )
   (/ x (boundp-else-bind-to-one y)))
```

Common Lisp

```
(defmacro boundp-else-bind-to-one (symbol)
          `(if (boundp (quote ,symbol))
               ,symbol
              (setf ,symbol 1)))

(defun boundp-else-bind-to-one (symbol)
         (if (boundp symbol)
              (symbol-value symbol)
             (setf (symbol-value symbol) 1)))
```


Logic-based example

my-theory: mortal(X) :- human(X).

Theories

john's-theory: human(X) :- greek(X).
greek(socrates).
P :- reflect(meta-t,P).

meta-t: provable(T,F) :- theorem(T,F)
provable(T,and(F,G)) :- provable(T,F), probable(T,G).
provable(T,F) :- clause(T,F,G), provable(T,G).
provable(T,F) :- clause(my-theory,F,G),provable(T,G),assert(theorem(T,F)).

Meta-theory
(provable, clause)

- John-theory mortal(socrates)
  - failure → reflect(meta-t, mortal(socrates))
- Meta-theory
  - provable(john-theory,mortal(socrates) →
    clause(my-theory,mortal(socrates),human(socrates)),
    provable(john-theory,human(socrates)),
    assert(theorem(john-theory,mortal(socrates)))

Rule-based example

Working memory: ((s1 . true) (s2 . true) (s3 . true))
Goal: ((p1 . true)(p2 . true))
Rules:
(1) if s1 and s2 then set(p1,true) and set(p2,false)
(2) if s3 then set(p2,true) and set(p1,false)
…
Meta-rules:
(3) if error-flag-1 then set(data-elm(s2),False)
and set(rule-to-be-fired(2), True)
(4) if satisfied(1) and satisfied(2) then set(error-flag-1,True)

- Reflection with the goal rule-to-be-fired(?rule) = true
- Contradiction: rules (1) and (2) → meta-rule (4)
- Error flag to true
- Rule (3) → s2 to false and the rule to be fired (2)

Object-based example

- 3-KRS (P. Maes)
- All are objects, data as well as programs
- Meta-objects
- Every object in the language has a meta-object

Implementation

- Common issue: all the languages operate by means of a meta-circular interpreter.
- Representation of the interpreter of the language (in the same language) is actually used to run the language.
- Minimum representation: a name for the interpreter (eval in LISP) plus some reified interpreter data (environment, continuation, ...).
- Procedural reflection: the causal connection is easy, the self-representation is used to implement the system.
- First example:
  - Interpreter program: eval
  - Program: expr
  - Environment: env
  - Continuation: cont

Scheme

```
(define meta-1 (expr &optional env cont)
               (eval expr env cont))
```


Extended version

Explicit internal aspects of Lisp

```
(define meta-circular-2 (expr &optional (env ()))
        (cond
          ((null expr) nil)
          ((numberp expr) expr)
          ((eq expr t) expr)
          ((symbolp expr) (binding expr env))
          ((eq (first expr) 'quote) (second expr))
          ((primitive-function-p (first expr))
            (apply (first expr)
                   (make-list-of-evaluated-args (cdr expr) env)))
          (t (eval (definition-of (first expr))
                   (lexic-meta
                     (definition-args-of (first expr))   env
                     (cdr expr)
                     env)))))
```

```
(define variant-meta-1 (expr &optional env  cont)))
        (do-something-with-the-result
          (eval (do-something-with-the-input expr) env cont)))
```

```
(loop (print (do-something-with-the-result
                (eval (do-something-with-the-input (read))))))
```

Common Lisp

```
(define lexic-meta (args-def args-for old-env)
          (cond ((null args-def) nil)
                (t (add-binding (first args-def)
                                (eval (first args-for) old-env)
                                (lexic-meta (cdr args-def)
                                            (cdr args-for)
                                            old-env)))))
```


- Variant that modifies the interpreter such that it has dynamical scoping
- Not possible with the first version

```
(define dinam-meta (args-def args-for old-env)
          (cond ((null args-def) old-env)
                (t (add-binding (first args-def)
                                (eval (first args-for) old-env)
                                (dina-meta (cdr args-def)
                                           (cdr args-for)
                                           old-env)))))
```

Lisp Example

- Roots of Lisp
- Primitives
  1. quote
  2. atom
  3. eq
  4. car
  5. cdr
  6. cons
  7. cond
- More
  1. null.
  2. and.
  3. not.
  4. append.
  5. pair.
  6. assoc.

Interpreter

```
(defun eval. (e a)
  (cond
   ((atom e) (assoc. e a))
   ((atom (car e))
    (cond
     ((eq (car e) 'quote) (cadr e))
     ((eq (car e) 'atom) (atom (eval. (cadr e) a)))
     ((eq (car e) 'eq) (eq (eval. (cadr e) a (eval. (caddr e) a)))
     ((eq (car e) 'car) (car (eval. (cadr e) a)))
     ((eq (car e) 'cdr) (cdr (eval. (cadr e) a)))
     ((eq (car e) 'cons) (cons (eval. (cadr e) a) (eval. (caddr e) a)))
     ((eq (car e) 'cond) (evcon. (cdr e) a))
     ('t (eval. (cons (assoc. (car e) a) (cdr e)) a))))
   ((eq (caar e) 'label)
    (eval. (cons (caddar e) (cdr e))
           (cons (list (cadar e) (car e)) a)))
   ((eq (caar e) 'lambda)
    (eval. (caddar e)
           (append. (pair. (cadar e) (evlis. (cdr e) a)) a)))))
```