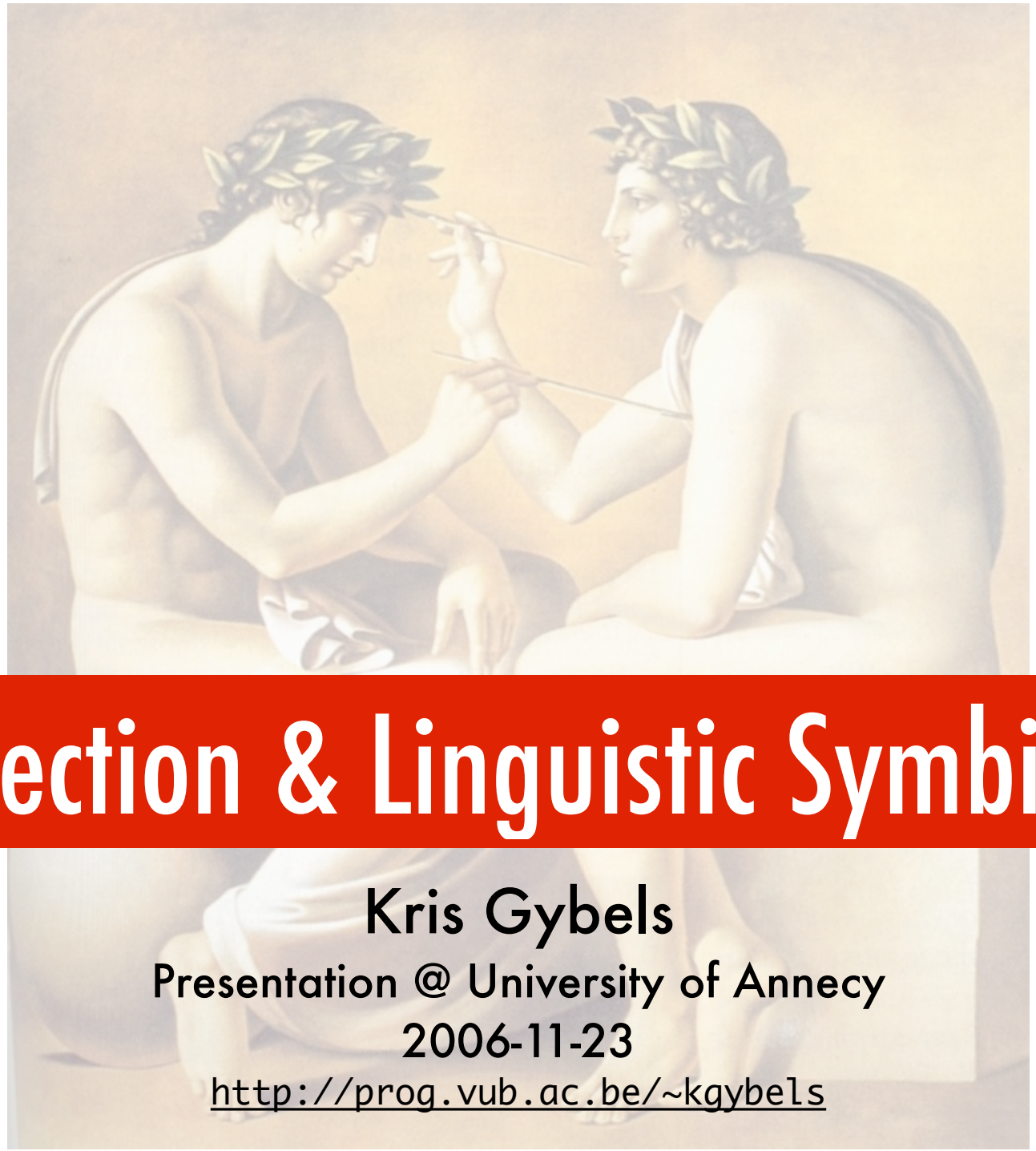


Reflection & Linguistic Symbiosis

Kris Gybels

Presentation @ University of Annecy

2006-11-23

http://prog.vub.ac.be/~kgybels

In the Exercises …

thisContext

SomeClass compile: 'bla ^ 5'

(Computational) Reflection

=

Ability for programs to reason about themselves

Overview

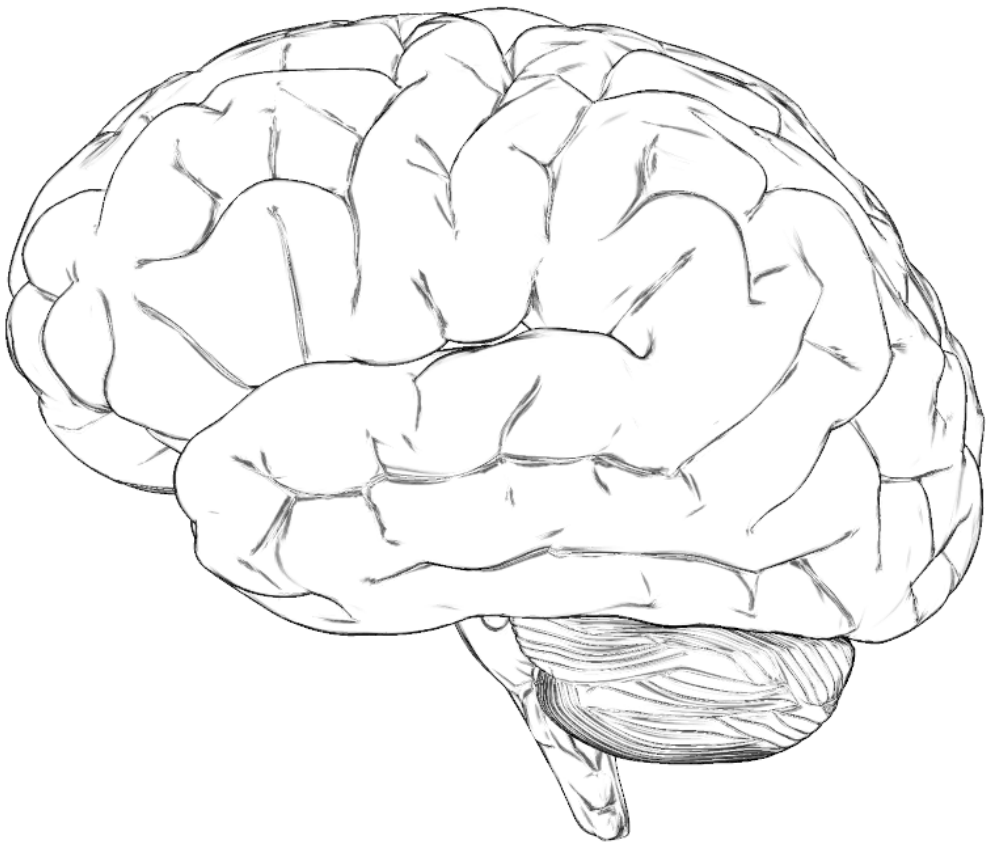
"Procedural Reflection in Programming Languages"
Brian C. Smith (1982)
http://hdl.handle.net/1721.1/15961

"Computational Reflection"
Pattie Maes (1987)
http://prog.vub.ac.be/Publications/1987/vub-arti-phd-87_2.pdf

"Open Design of Object-Oriented Languages"
Patrick Steyaert (1994)
http://prog.vub.ac.be/Publications/1994/vub-prog-phd-94-01/

The Maes View

Reflection & Artificial Intelligence



LEX2

S: ∫ 3x cos(x) dx --> Apply substitution
with u = 3x, and dv = cos(x) dx

G: ∫ f1(x) f2(x) dx --> Apply substitution
with u = f1(x), and dv = f2(x) dx

LEX2

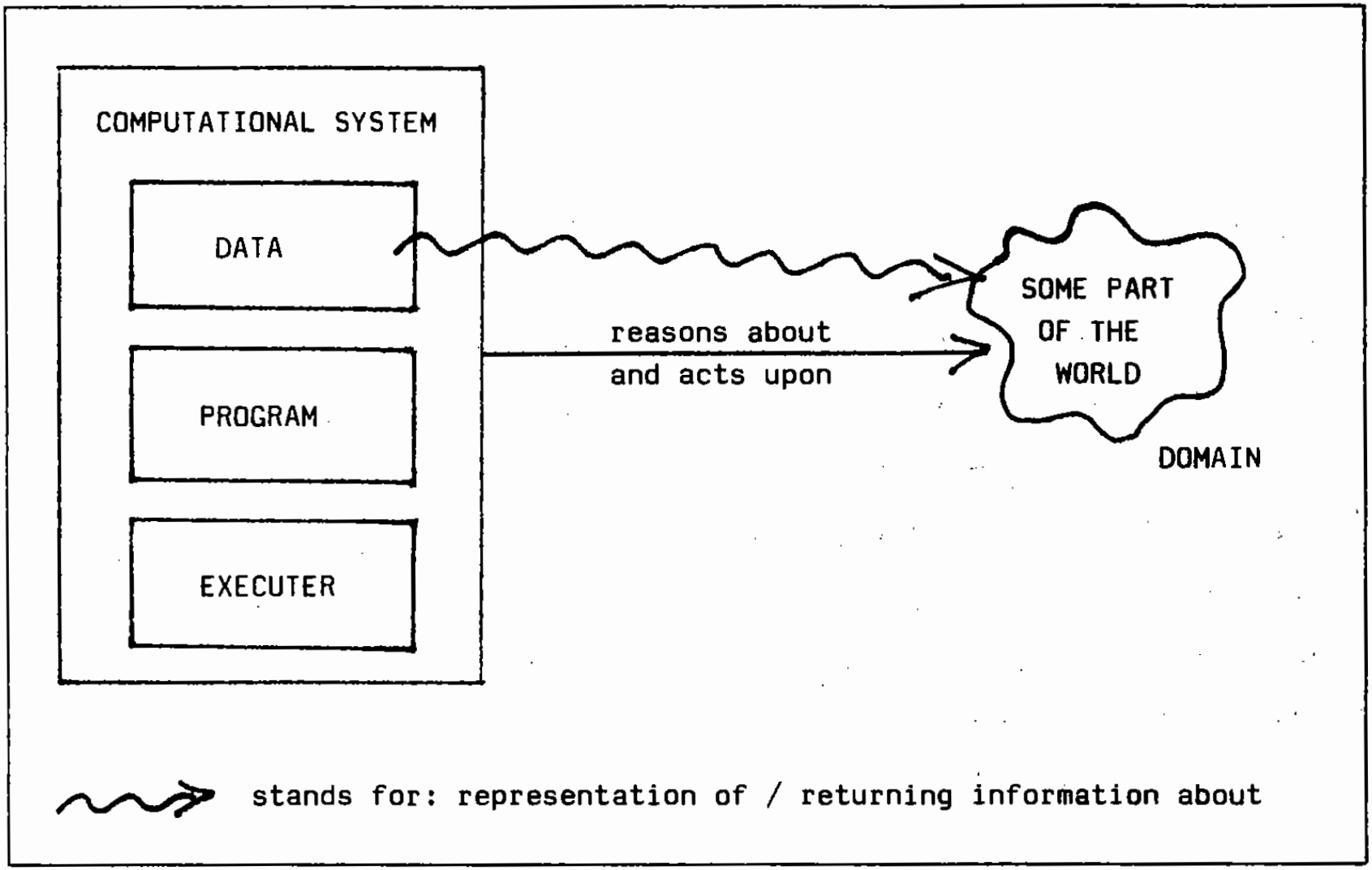
```
S: ∫ 3x cos(x) dx --> Apply substitution
                          with u = 3x, and dv = cos(x) dx

G: ∫ f1(x) f2(x) dx --> Apply substitution
                            with u = f1(x), and dv = f2(x) dx
```

```
S: ∫ 3x trig(x) dx --> Apply substitution
                           with u = 3x, and dv = trig(x) dx

G1: ∫ poly (x) f2(x) dx --> Apply substitution
                                with u = poly(x), and dv = f2(x) dx

G2: ∫ f1(x) transc(x) dx --> Apply substitution
                                 with u = f1(x), and dv = transc(x) dx
```

MYCIN

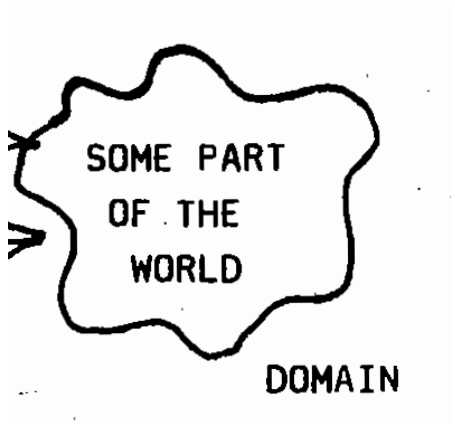
(a) IF it is not known whether there are factors that interfere with
the patients bleeding,
THEN it is definite (1.0) that there are no factors that interfere
with the patients bleeding.

Some Terminology



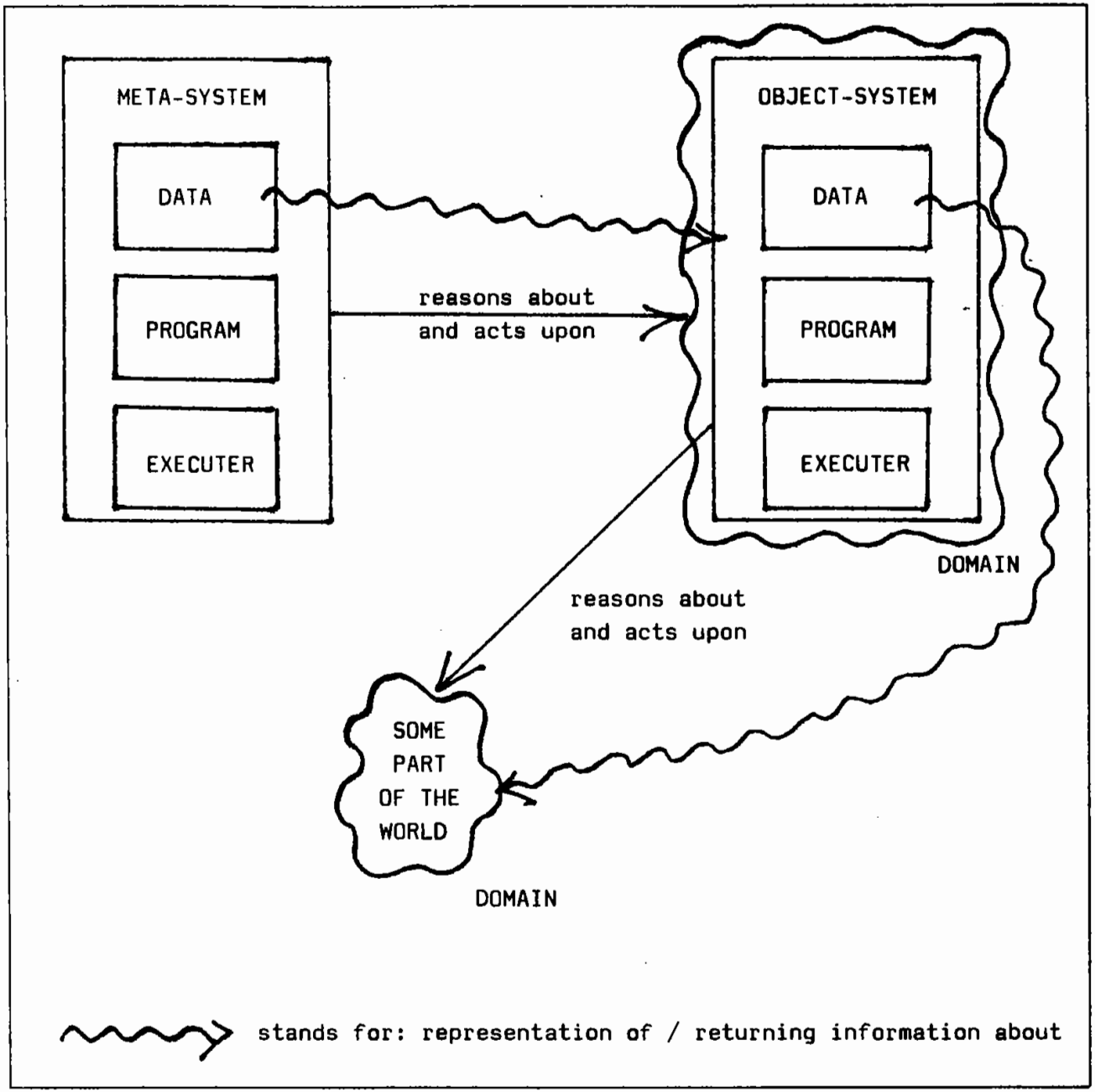
Programming


```
changeColor: c
  color := c
nextColor
  (color = #green) ifTrue: [ ... ]
  ...
```



a TrafficLight	
color	

Meta Programming

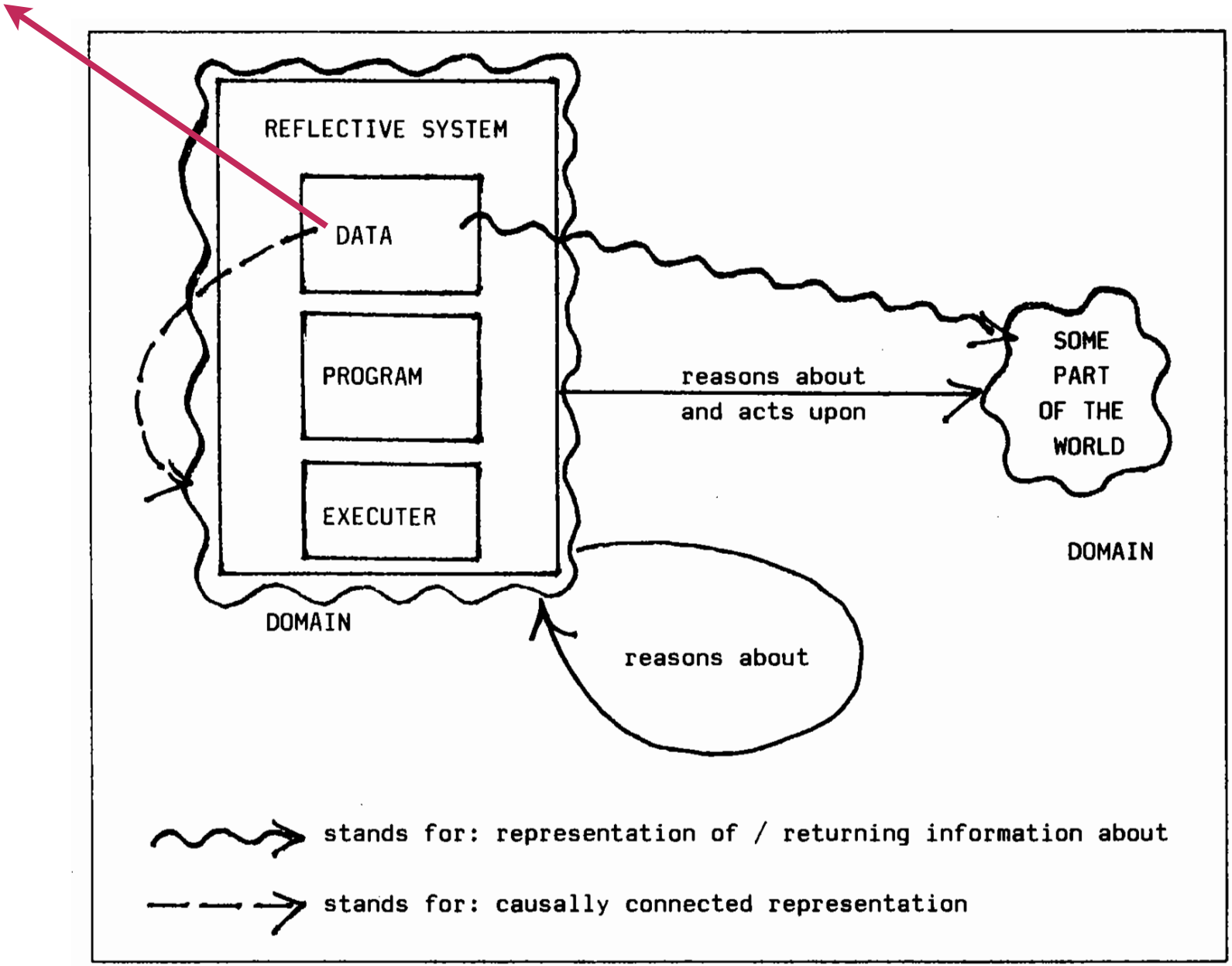


e.g. an executer (interpreter) (!)

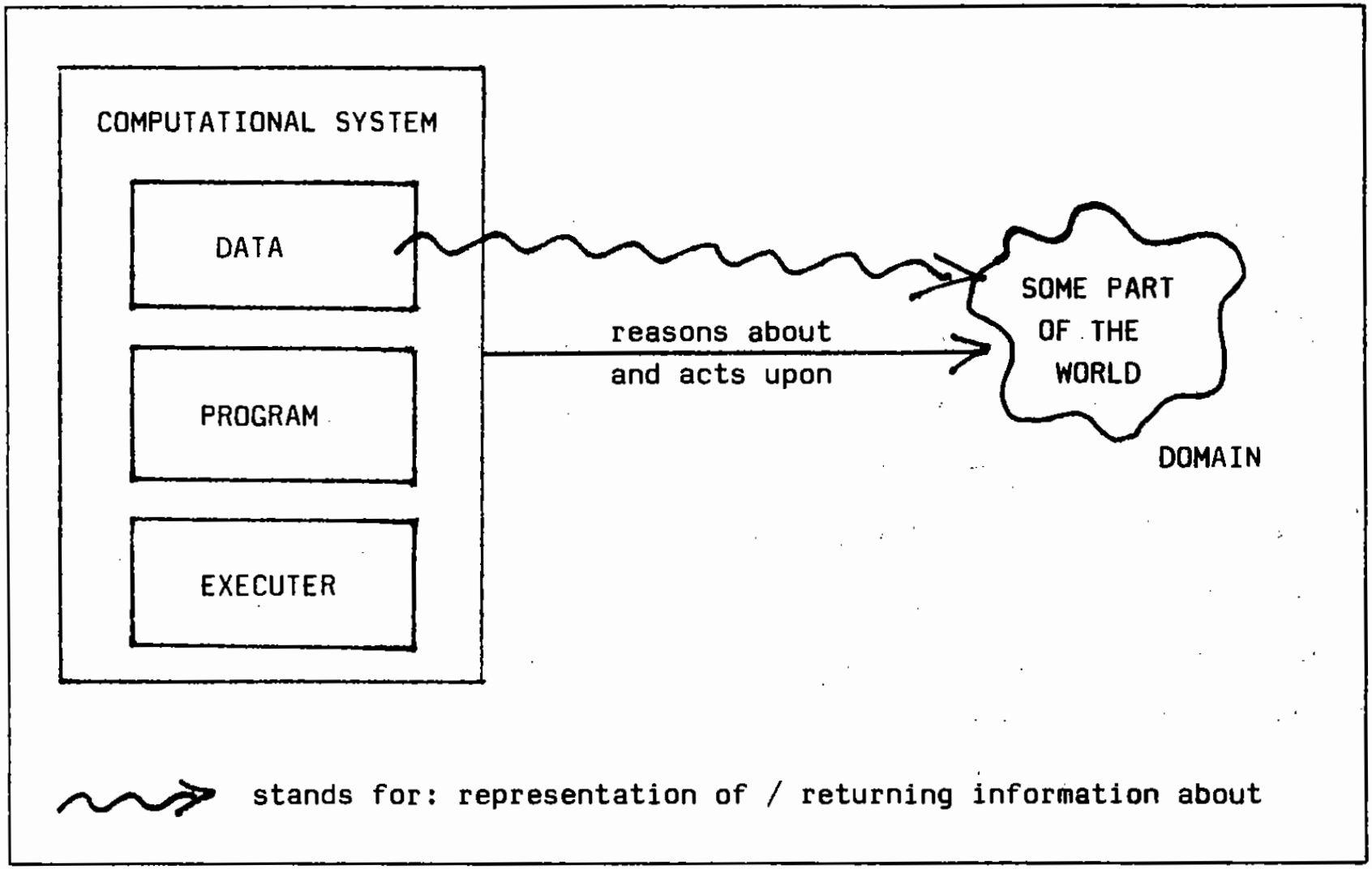
a debugger

Reflective Systems

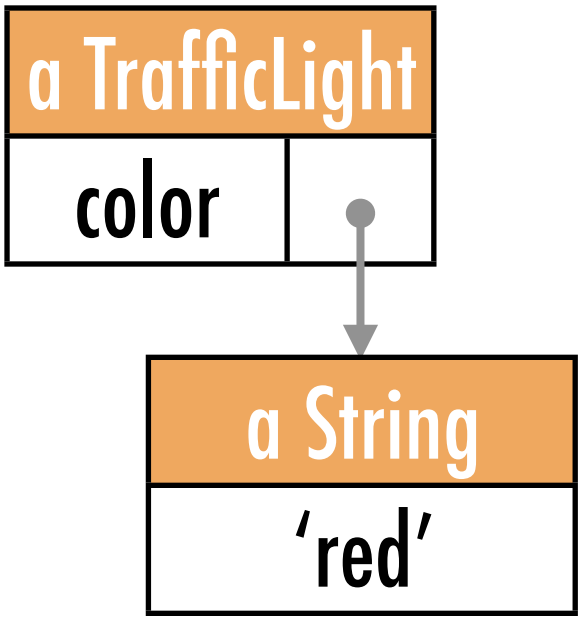
self representation



Causal Connection



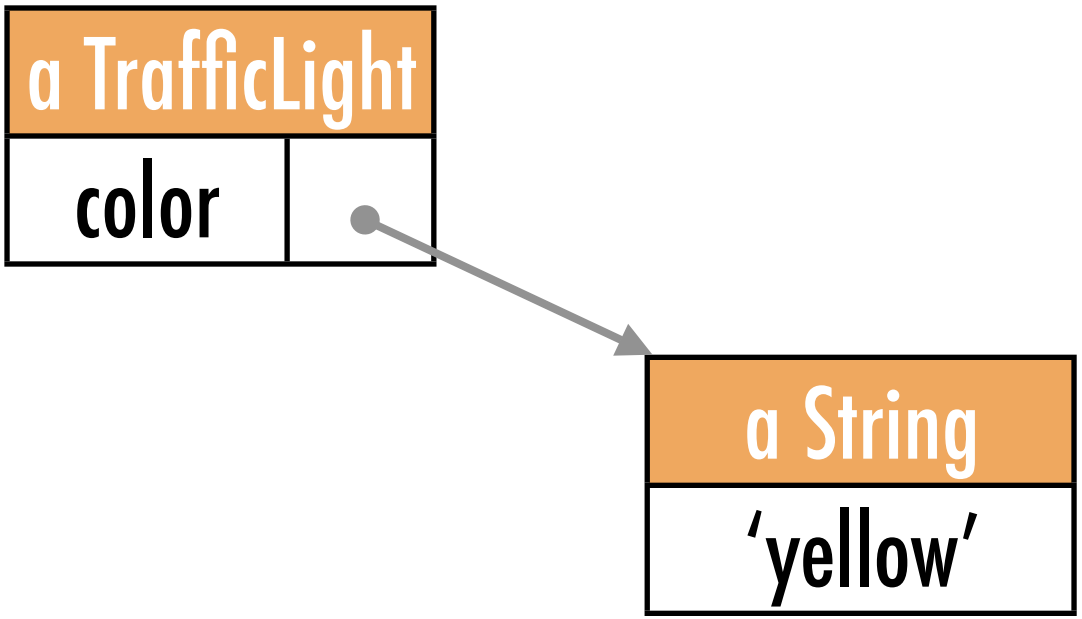
Causal Connection



Causal Connection

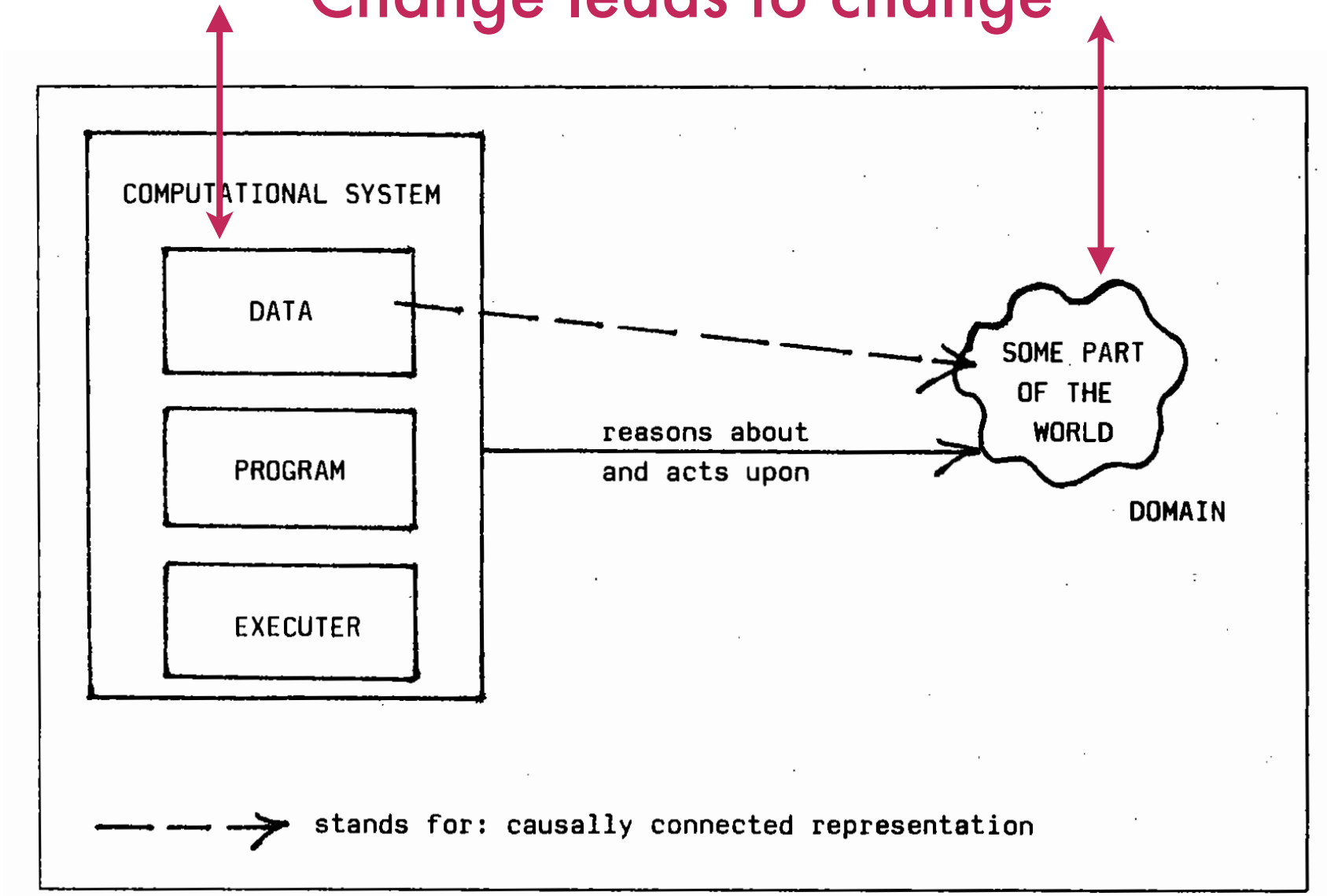
a TrafficLight	
color	

Causal Connection

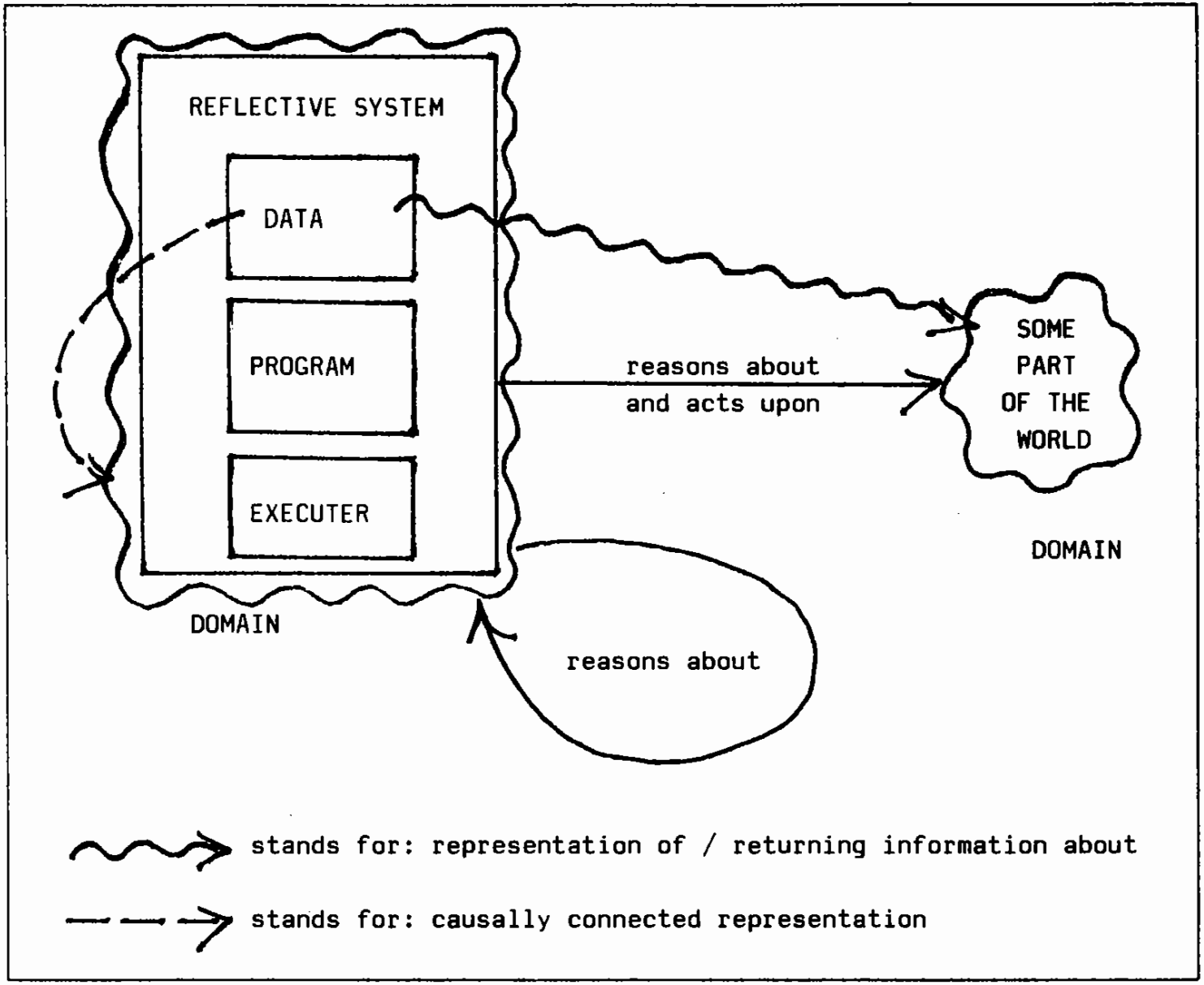


Causal Connection

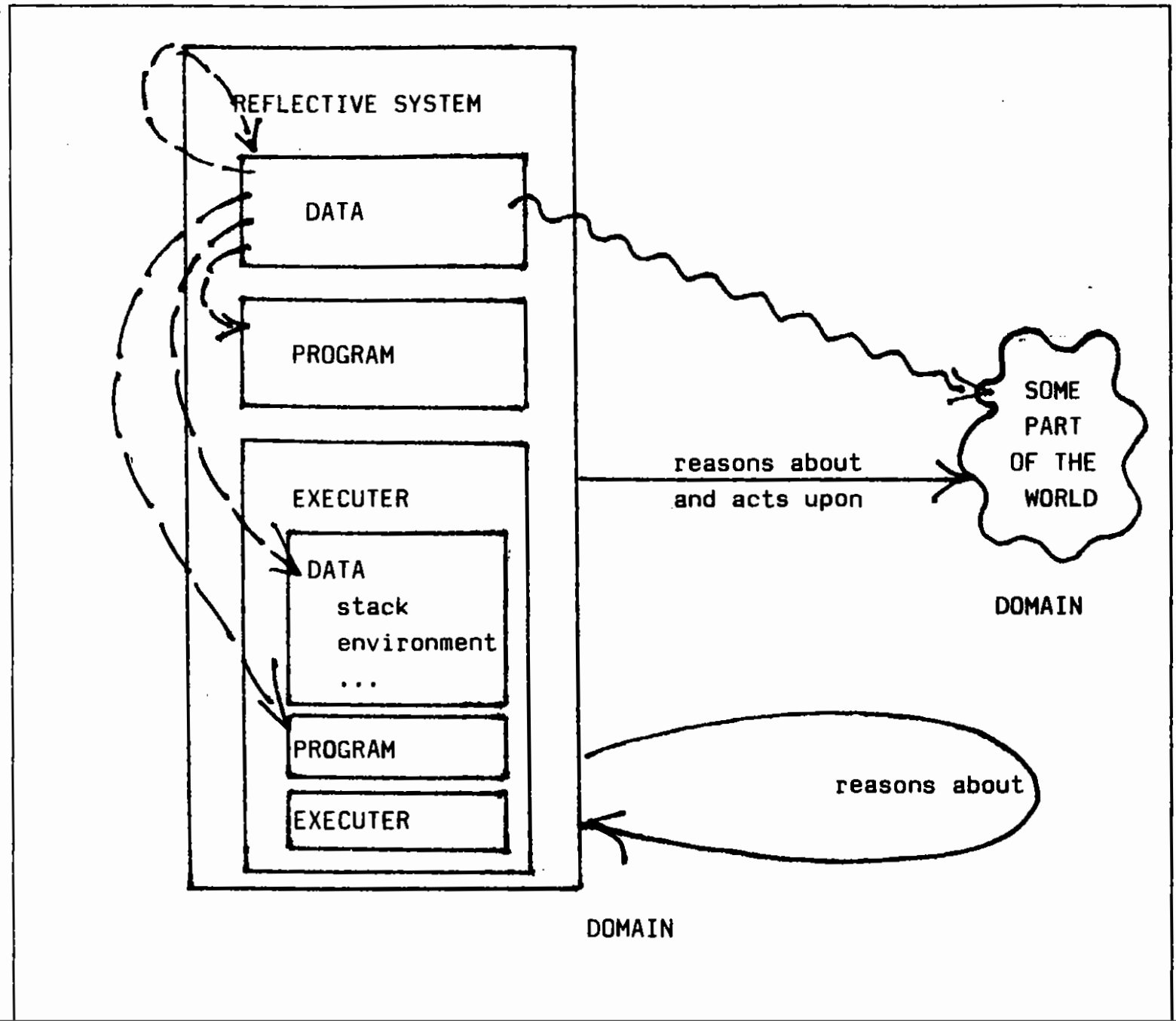
Change leads to change



Self Representation?



The Executer & Possible Self Reps



Importance of CC for Reflection

```
test1
  self test2.
  Transcript show: 'Called test2';cr.
```

```
test2
  self test3.
  Transcript show: 'Called test3';cr.
```

```
test3
  thisContext sender: thisContext sender sender.
```

The Smith View aka 3-LISP Very Briefly

How the CC is achieved in 3-LISP

test3
thisContext sender receiver

```
(define BOUND
   (lambda reflect [[var] env cont]
      (if (bound-in-env var env)
          (cont '$T)
          (cont '$F))))
```

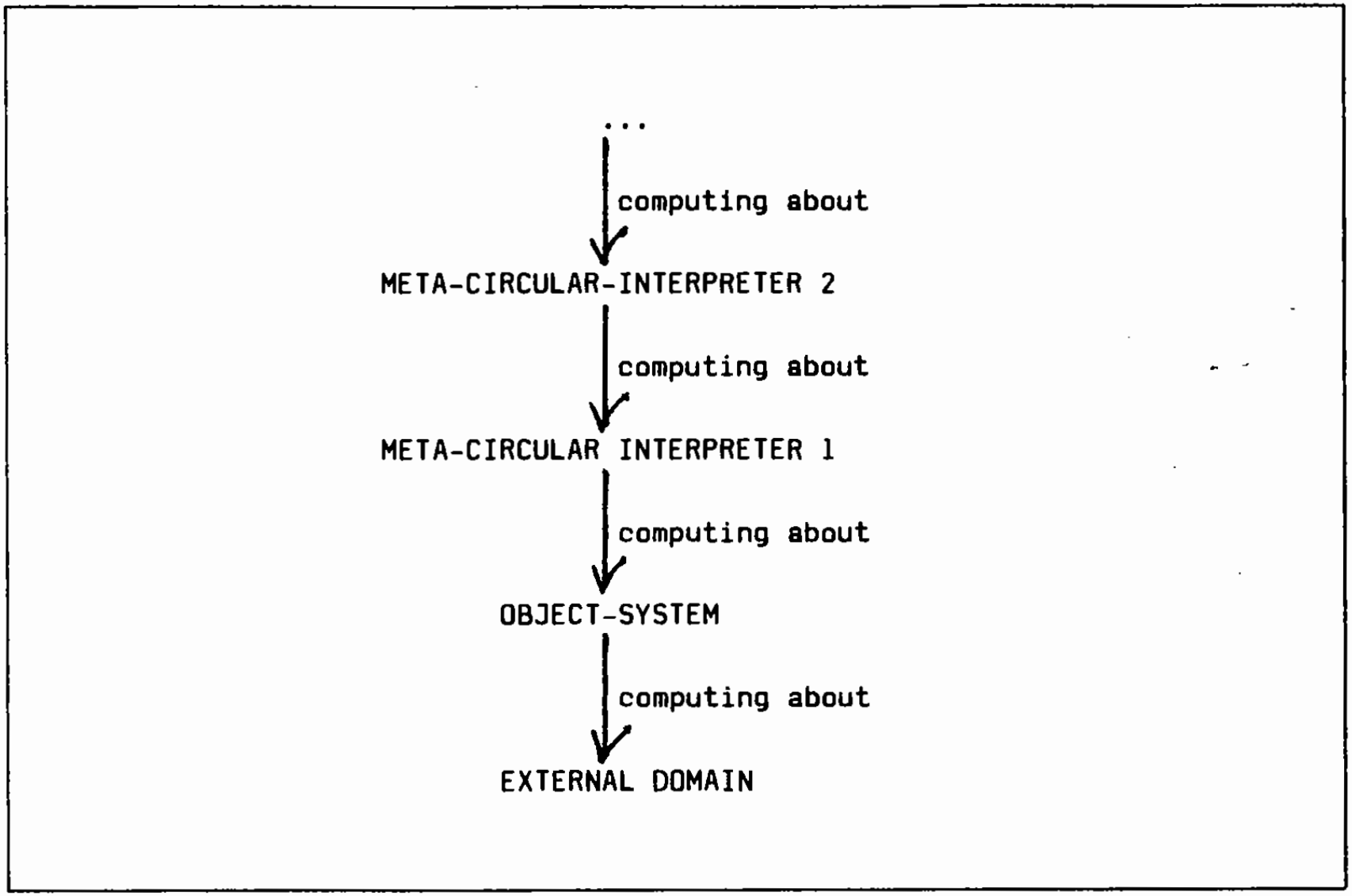
Reflective code should be run at the same level as the meta-circular processor; it should not be processed by the meta-circular processor.

3-LISP Very Briefly

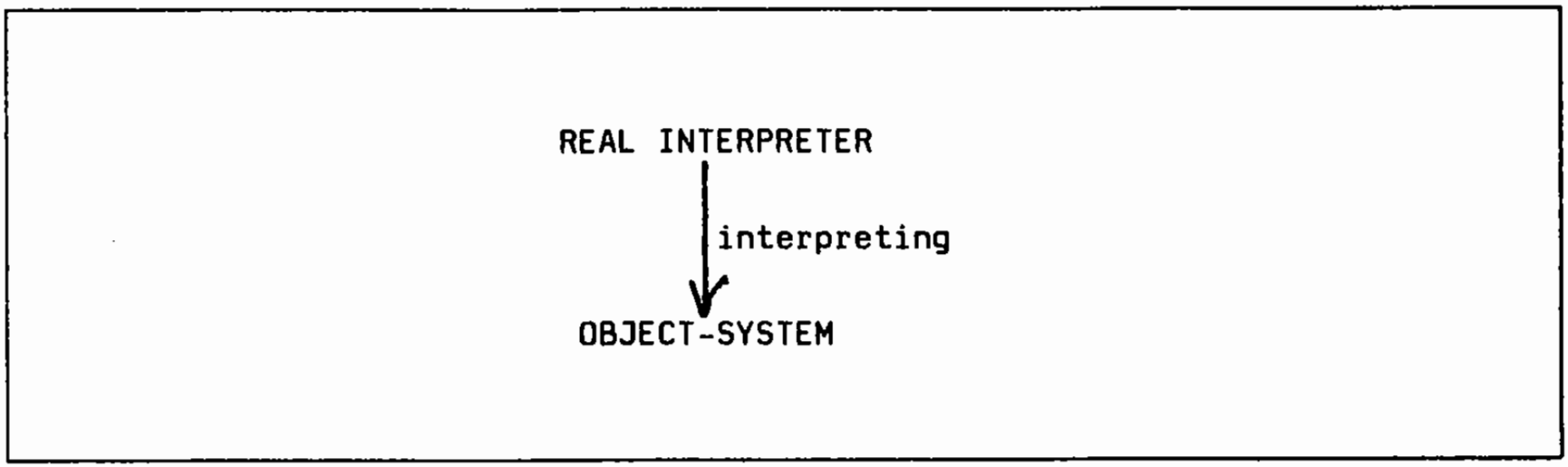
```
(define BOUND
    (lambda reflect [[var] env cont]
       (if (bound-in-env var env)
           (cont '$T)
           (cont '$F))))
```

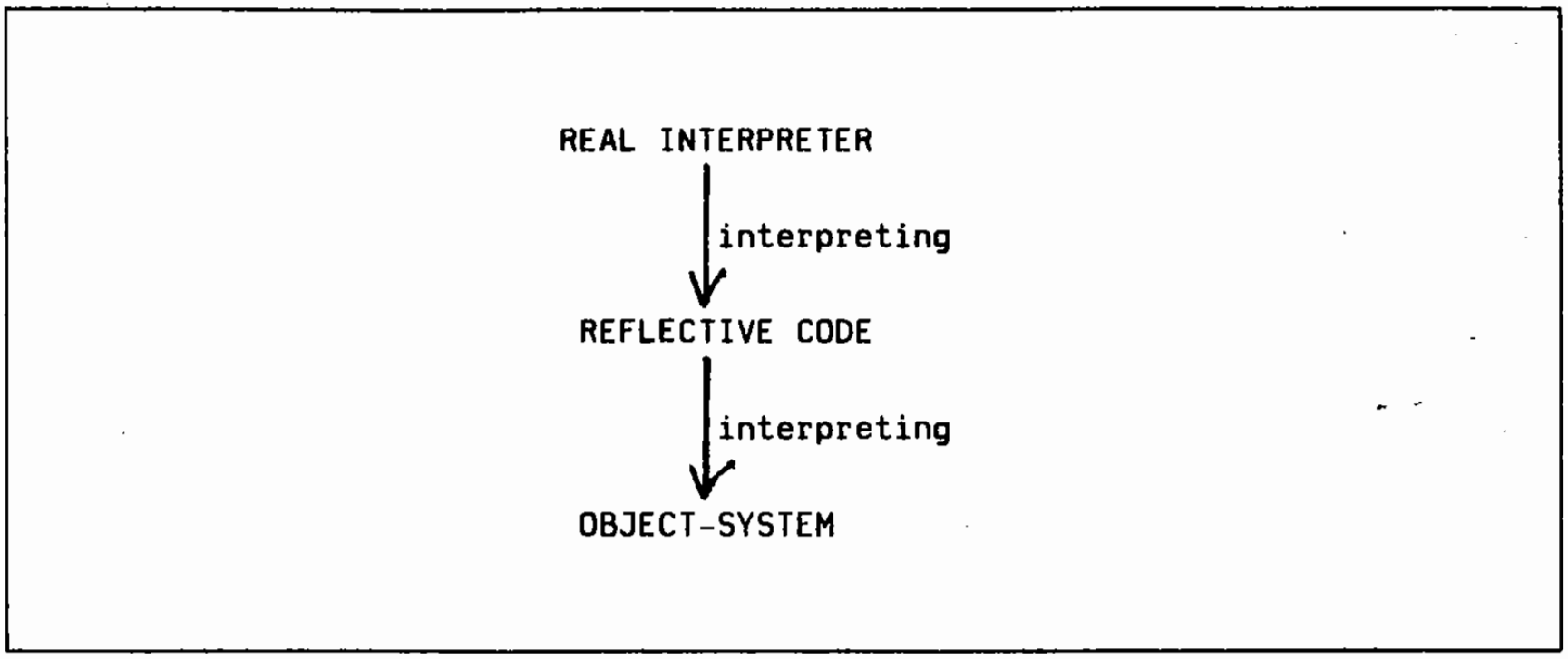
```
(LET [[X 3]] (BOUND X)) ⇒ $T,

(BOUND X) ⇒ $F
```

3-LISP Reflective Tower



Finite Realization of 3-LISP Tower





Steyaert: Agora & Linguistic Symbiosis

Agora vs. 3-LISP

Code written in the object-system can be executed in the meta-system

Language real interpreter != Language object-system

Agora Solution:

Define linguistic symbiosis

3-LISP Solution:

Insert meta-circular 3-LISP

Agora

Framework for Object-Oriented Language

+

Symbiosis Ability

=

Language with Support for Reflection

!! Implementation language must be object-oriented !!

Implementations with Linguistic Symbiosis of Agora in Smalltalk, C++ & Java

Agora Example (1)

Agora: an example

```
[ Self makeComplex
        μmethod:
            [ realP μvar: 0.0;
              imagP μvar: 0.0;
              self real: realA imag: imagA
                    μmethod:
                        [ ((self μclone) real: realA) imag: imagA ];
              self real: arg
                    μmethod:
                        [ realP μ<- arg;
                          self ];
              self imag: arg
                    μmethod:
                        [ imagP μ<- arg;
                          self ];
              self real
                    μmethod:
                        [ realP ];
              self imag
                    μmethod:
                        [ imagP ];
```

a Complex mixin

instance variables

instance methods

Agora Example (2)

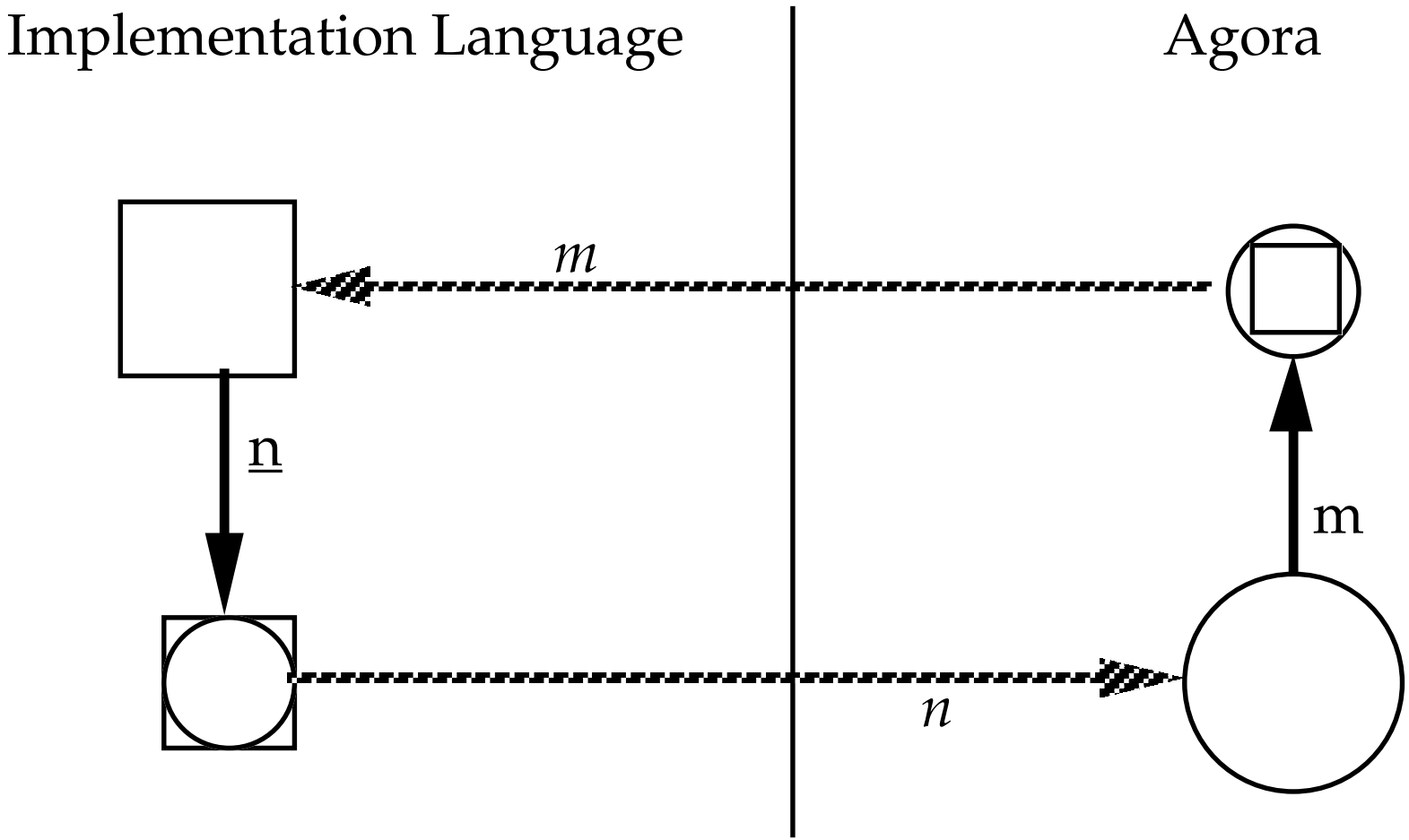
Agora: an example

```
                self mod
                        μmethod:
                            [ (realP sqr + imagP sqr) sqrt ];
                self + arg
                        μmethod:
                            [ self real: (arg real + realP)
                                    imag: (arg imag + imagP) ];
                μself];

Complex μvar: Object makeComplex;

c1 μvar: Complex real: 1.5 imag: 2.0;
c2 μvar: Complex real:-2.0 imag: 1.5;
c3 μvar: c1 + c2;
"real part = " display;
(c3 real) display;
eoln display;
"imag part = " display;
(c3 imag) display;
eoln display] μvalue
```

instance methods

instantiation

Linguistic Symbiosis Concept



Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";

frame addComponent: ok;

okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];

ok addActionListenerActionListener: okListener
```

Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";

frame addComponent: ok;

okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];

ok addActionListenerActionListener: okListener
```

Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";
```

```
frame addComponent: ok;
```

```
okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];

ok addActionListenerActionListener: okListener
```

Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";
```

```
frame addComponent: ok;
```

```
okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];
```

```
ok addActionListenerActionListener: okListener
```

Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";
```

```
frame addComponent: ok;
```

```
okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];
```

```
ok addActionListenerActionListener: okListener
```

Non-Reflective Example

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";
```

```
frame addComponent: ok;
```

```
okListener μVARIABLE: [
  implements μMETHOD:
     (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
     ("java.lang.Object" μJAVA);
  actionPerformedActionEvent: e μMETHOD: {
    ("java.lang.System" μJAVA) out printlnString: "Button Pressed!";
    frame setVisibleboolean: false
    }
  ];
```

```
ok addActionListenerActionListener: okListener
```

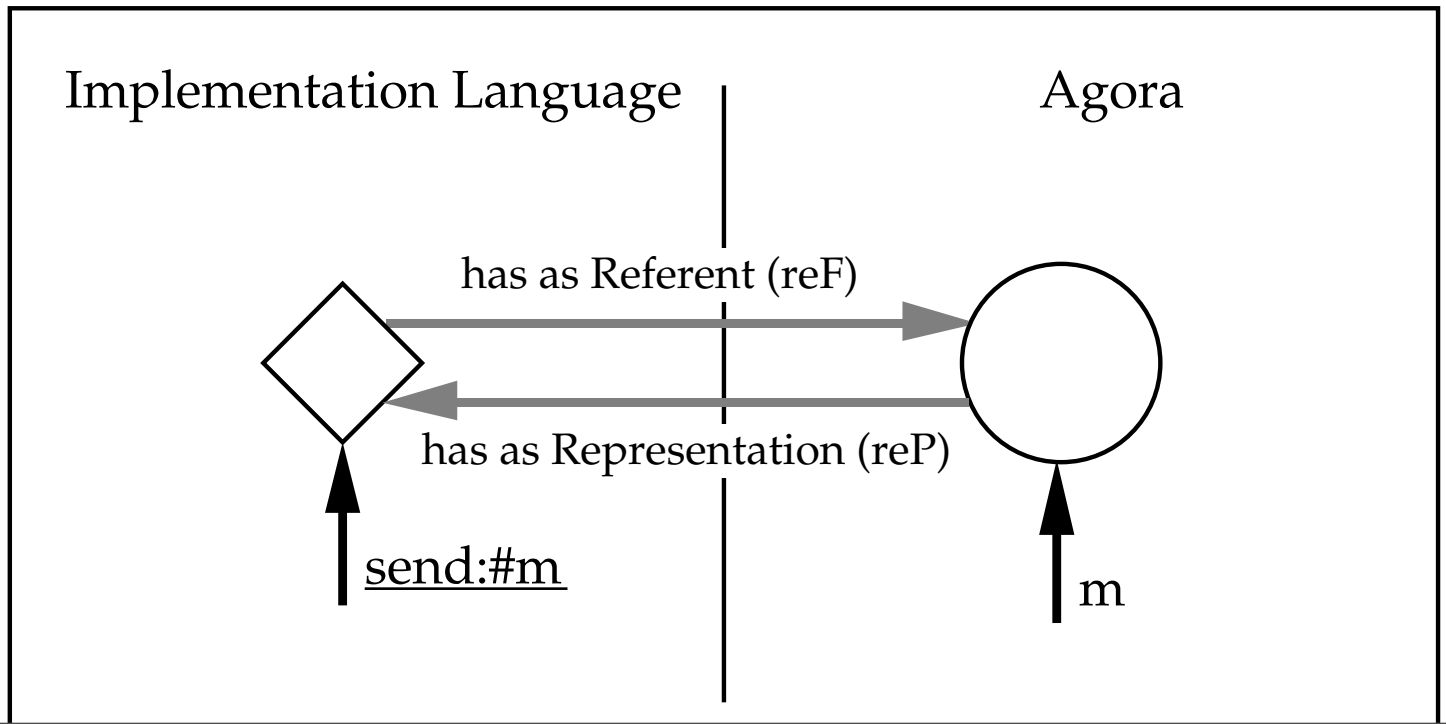
The Agora Framework: Objects

Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
    abstract send:pattern client:client result AbstractMetaObject
endclass
```

meta-system

object-system



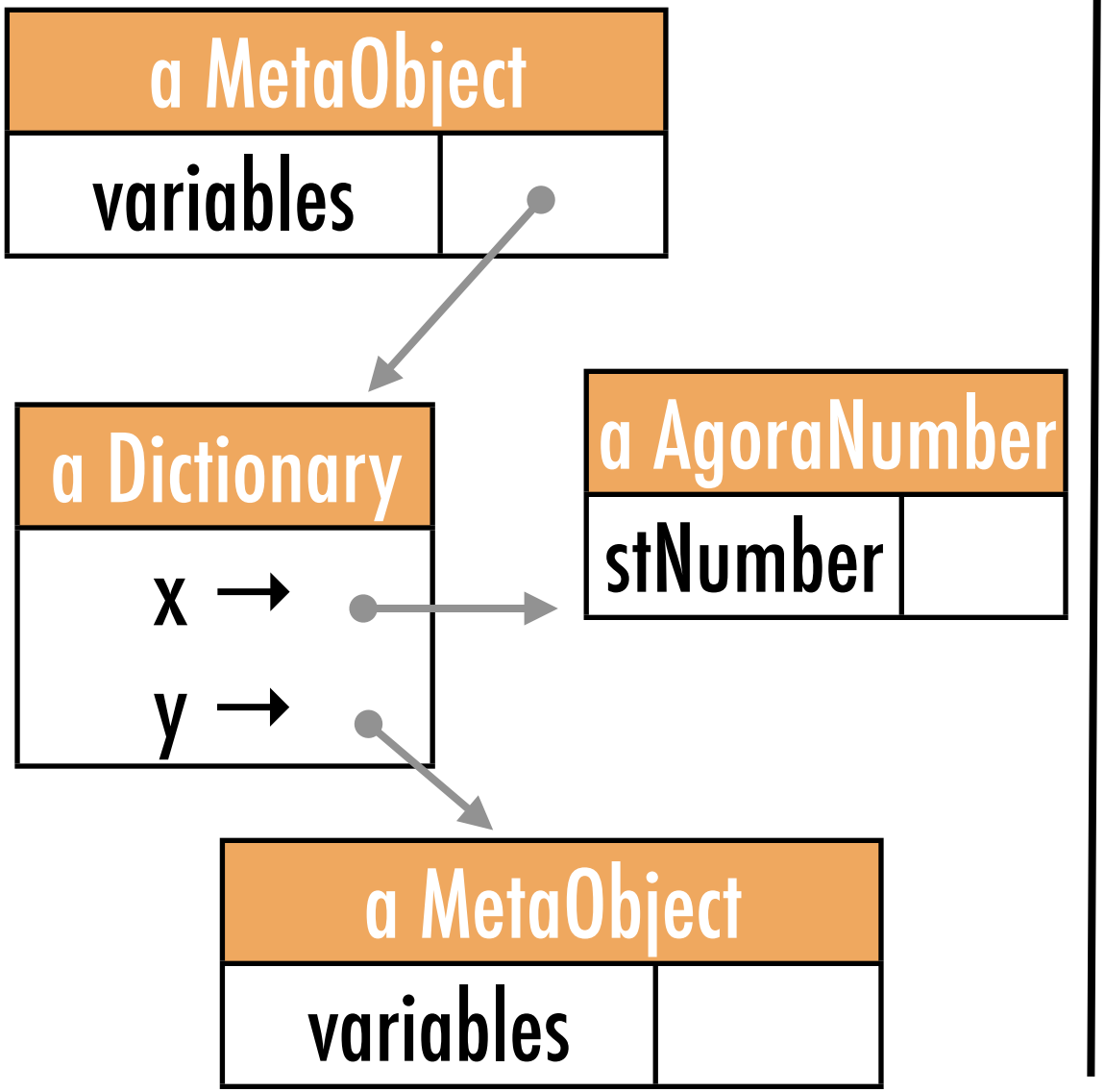
What's in a Meta-Object?

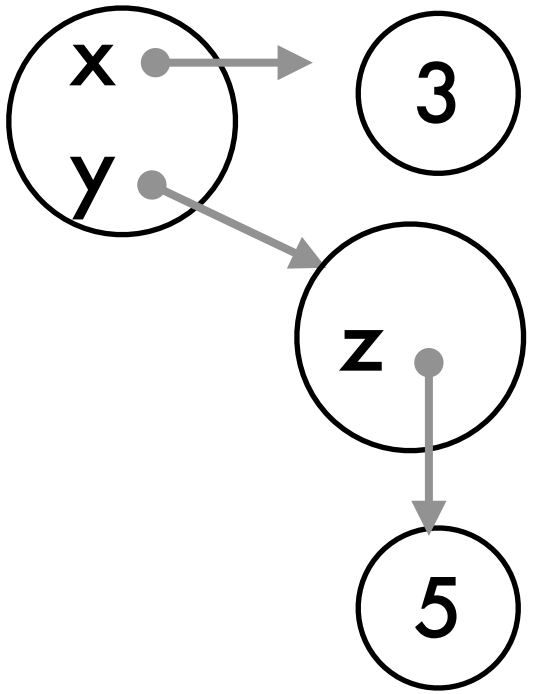
```
[
  x μvar: 3;
  y μvar: [ z μvar: 5 ]
]
```

What's in a Meta-Object?

meta-system

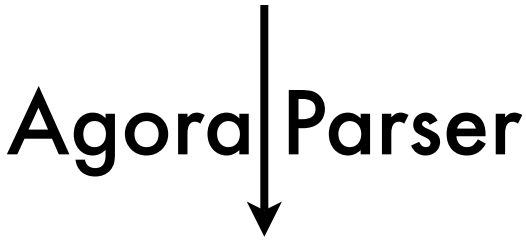
object-system





Agora Expressions

"frame addComponent: ok;"



Abstract Class for Expression Objects

```
class AbstractExpression
  methods
    abstract eval:context result AbstractMetaObject
endclass
```

eval: and send:client:

Agora Message Passing

```
class MessageExpression extends AbstractExpression
  instance variables
   receiver:AbstractExpression,
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandarContext+)
       result AbstractMetaObject
     local variables arguments:ArgumentList

     for each argument in pattern do
        arguments add:(argument eval:(context asFunctionalContext))

     ^(receiver eval:(context asFunctionalContext))
         send:(pattern asCategory:context)
         client:(StandardClient arguments:arguments)
endclass
```

Key Point in Implementing Symbiosis

```
frame μVARIABLE: ("java.awt.Frame" μJAVA) new;

ok μVARIABLE: ("java.awt.Button" μJAVA) newString: "OK";

frame addComponent: ok;
```

```
^(receiver eval:(context asFunctionalContext))
    send:(pattern asCategory:context)
    client:(StandardClient arguments:arguments)
```

Some Agora Terminology

	Implementation Language	Agora
referable	Implicitly Referable Object	Explicitly Referable Object
encoded	Implicitly Encoded Object	Explicitly Encoded Object
referable & encoded	Implicit Object	Explicit Object

Making Explicitly Referable

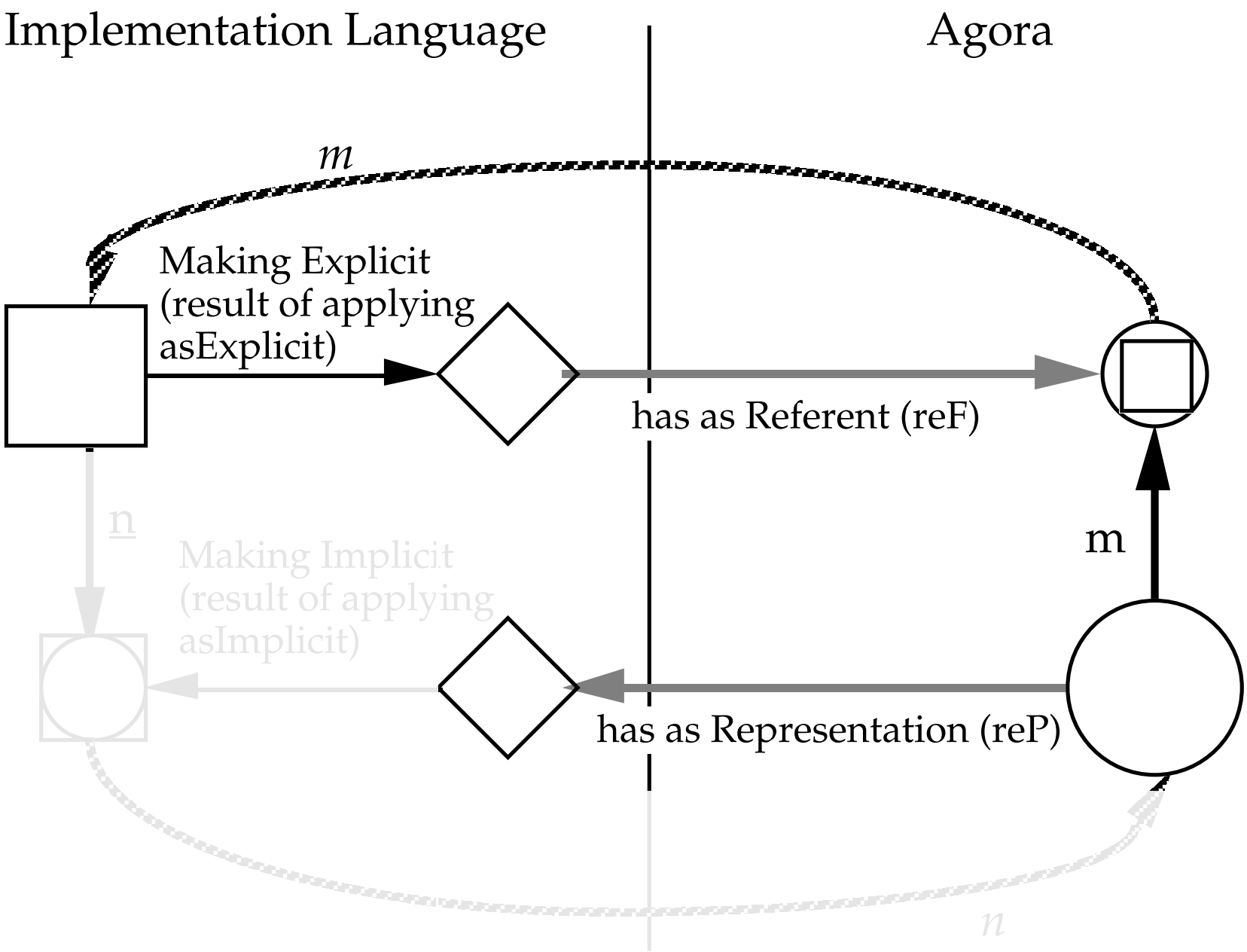
Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
    abstract send:pattern client:client result AbstractMetaObject
endclass
```

ExplicitlyReferableOrderedCollection

```
send: pattern client: client
  (pattern name = 'add:') ifTrue: [
     ^ (c add: (client arguments first asImplicit)) asExplicit
  ]
  (pattern name = 'size') ifTrue: [
    ^ (c size) asExplicit
  ]
```

Conversion Schema



Making Implicitly Referable

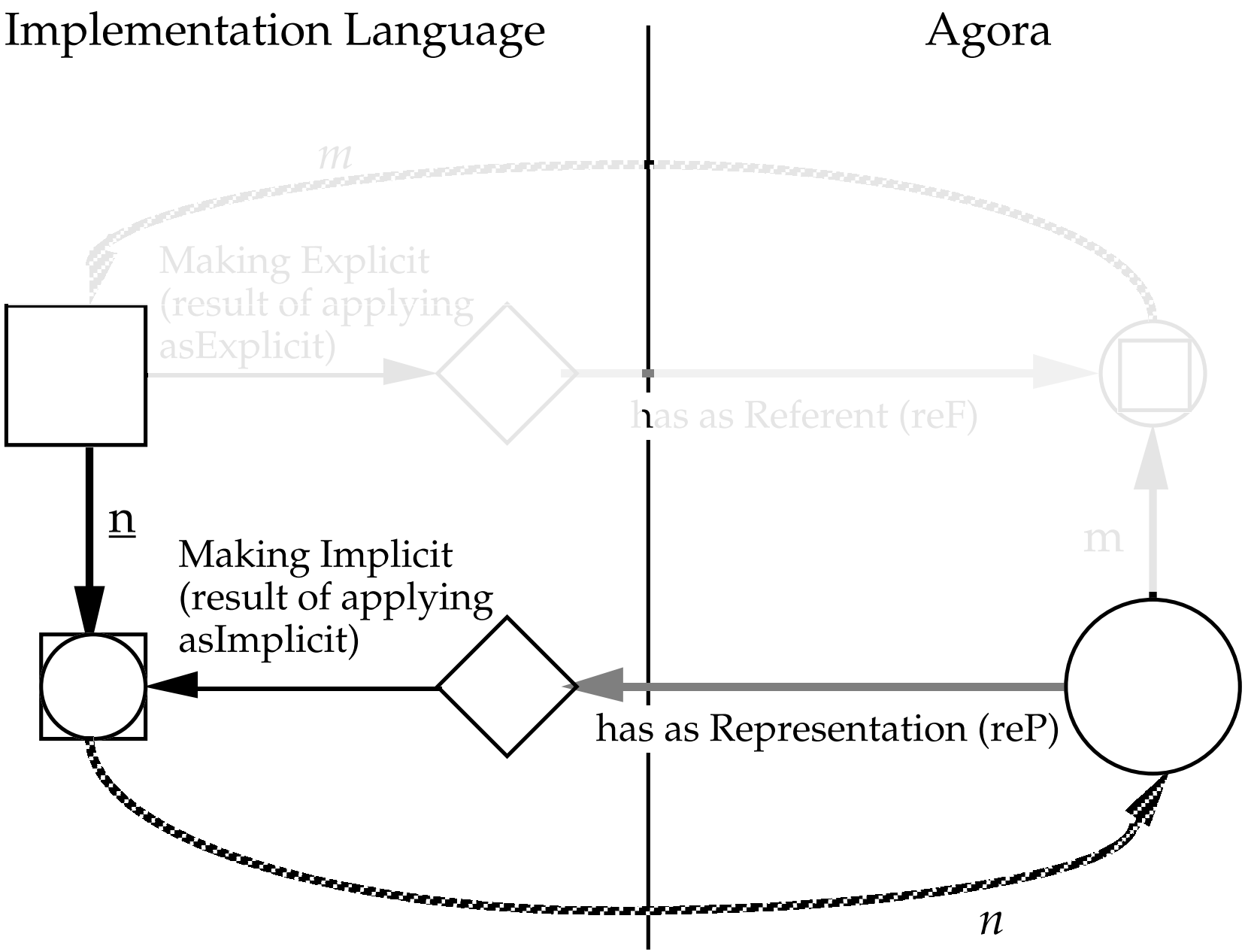
ImplicitlyReferableMetaObject

```
hash
  ^ (metaObj send: (“hash” makePattern)
              client: (EmptyClient new)) asImplicit

= otherSmalltalkObject
  ^ (metaObj send: (“=”) makePattern)
              client: (StdClient argument: otherSTObj)) asImplicit

test: obj1 with: obj2
  ^ ...

...
```

Smalltalk Dictionary Symbiosis Ex.

```
x:x y:y mixin method:
 [ x μmethod: x ;
   y μmethod: y ;
   hash μmethod: x + y ;
   = p μmethod: (x = p x) & (y = p y) ;
   print μmethod:[ x print ;y print ] ] ;

d variable: (“Dictionary” μSmalltalk) new ;
d at:(self x:10 y:20) put:"Wim" ;
d at:(self x:20 y:10) put:"Patrick" ;
d at:(self x:17 y:40) put:"Koen" ;
```

Conversion Schema



Using Symbiosis For Reflection

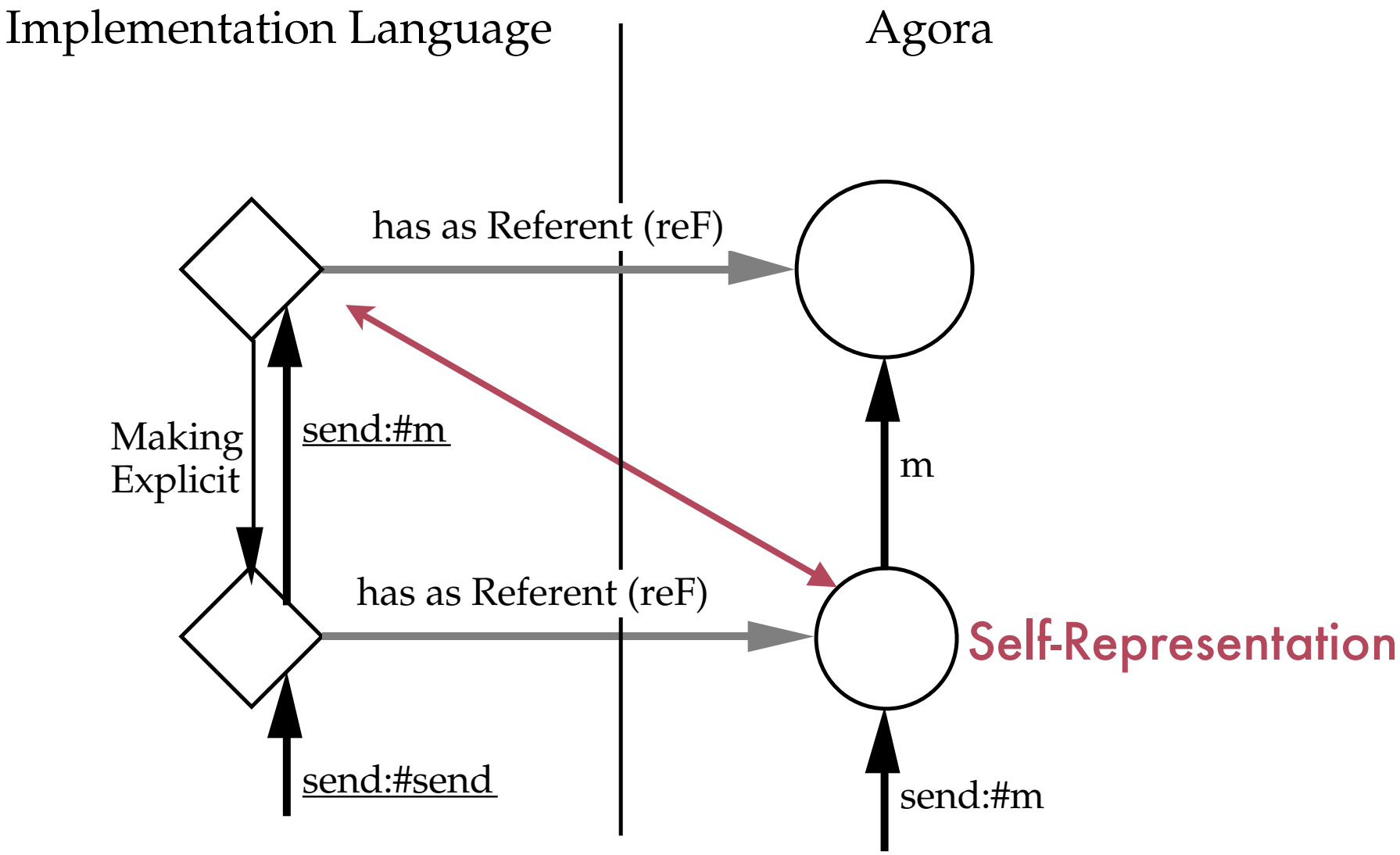
```
UnaryPattern define: … ;        —— a pattern prototype
EmptyClient define: … ;         —— a client prototype
Point define: … ;               —— a point prototype

metaOfP define ;
p define: Point x:3 y:4;
metaOfP  <- (p asMeta) ;
result <- metaOfP send:(UnaryPattern name:"x")
                    client:(EmptyClient clone)
```

Making Meta Objects Explicitly Ref.

Explicitly Referable Meta-objects

```
class ExplicitlyReferableMetaObject extends AbstractMetaObject
  constants
    SendPattern = KeywordPattern name:"send:client:"
  instance variables
    aMeta:AbstractMetaObject
  methods
    concrete send:pattern client:client result AbstractMetaObject
      if pattern = SendPattern
        then
          ^(aMeta
              send:(client firstArgument asImplicit)
              client:(client scndArgument asImplicit))asExplicit
        else
          ... raise an error
endclass
```

The Causal Connection



Defining Meta-Objects from Agora

```
MakeResultHolder Mixin:
  [ receiver define ; pattern define ; client define ;
    receiver:r pattern:p client:c CloningMethod:
      [ receiver <- r ; pattern <- p ; client <- c ] ;
    send:p client:c Method:
      [ (ResultHolder
            receiver: (receiver send:pattern client:client)
            pattern:p
            client:c) return ]
  ] ;
ResultHolder define: Object MakeResultHolder ;

MakeLazy Mixin:
  [ who define ;
    who:w CloningMethod: [ who <- w ] ;
    send:p client:c Method:
        [ (ResultHolder receiver:who pattern:p client:c) return ]
  ] ;
Lazy define: Object MakeLazy ;

Point define: … ;

p define: (Lazy who:((Point x:3 y:4) asMeta)) asObject ;
—— p contains a lazy point now
```

Issues in Symbiosis for Agora

Paradigm Differences
Agora vs. Implementation Language

Agora
- Prototype-Based
- Dynamically Typed

Smalltalk
- Class-Based
- Dynamically Typed

Java/C++
- Class-Based
- Statically Typed

Agora vs. Java

```
class MyCollection {
  /* ... */
  public void add(ClassA x) { ... }
  public void add(ClassB x) { ... }
  public void add(ClassC x) { ... }
}
```

```
someAgoraObject μvariable: ...
aJavaCollection μvariable: (“MyCollection” μJAVA) new

aJavaCollection add: someAgoraObject
```

Agora in Symbiosis with Java

Explicitly Referable Implicitly Encoded Objects

```
x addComponent: y string: z
```

invokes method “add” which takes a Component and a String as argument

Making Explicit Objects Implicit

```
okListener μVARIABLE: [
  implements μMETHOD:
    (1 μARRAY: ("java.awt.event.ActionListener" μJAVA));
  replaces μMETHOD:
    ("java.lang.Object" μJAVA);
  ...
]
```

SOUL

Symbiosis of SOUL and Smalltalk

! Even Bigger Paradigm Differences !

Logic Programming vs Object-Oriented Programming

- Logic rules can ‘return’ multiple results
- Logic rules can be called ‘unbound’ parameters

New Interpretation Process

```
?list containsSomethingRed if
  ?list contains: ?x &
  ?x isRed
```

New Interpretation Process

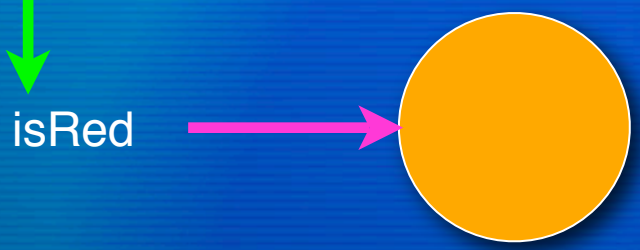
```
?list containsSomethingRed if
  ?list contains: ?x &
  ?x isRed
```

No rule for isRed, so send as message to object in ?x

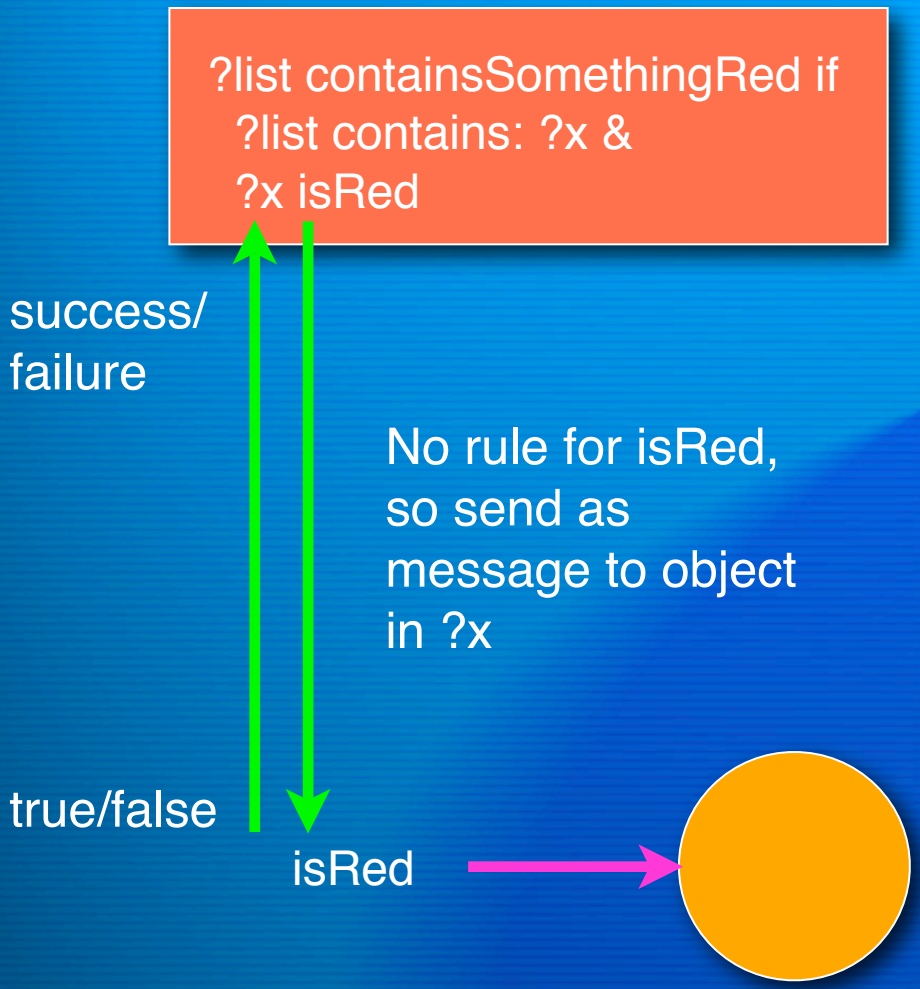
New Interpretation Process

```
?list containsSomethingRed if
  ?list contains: ?x &
  ?x isRed
```

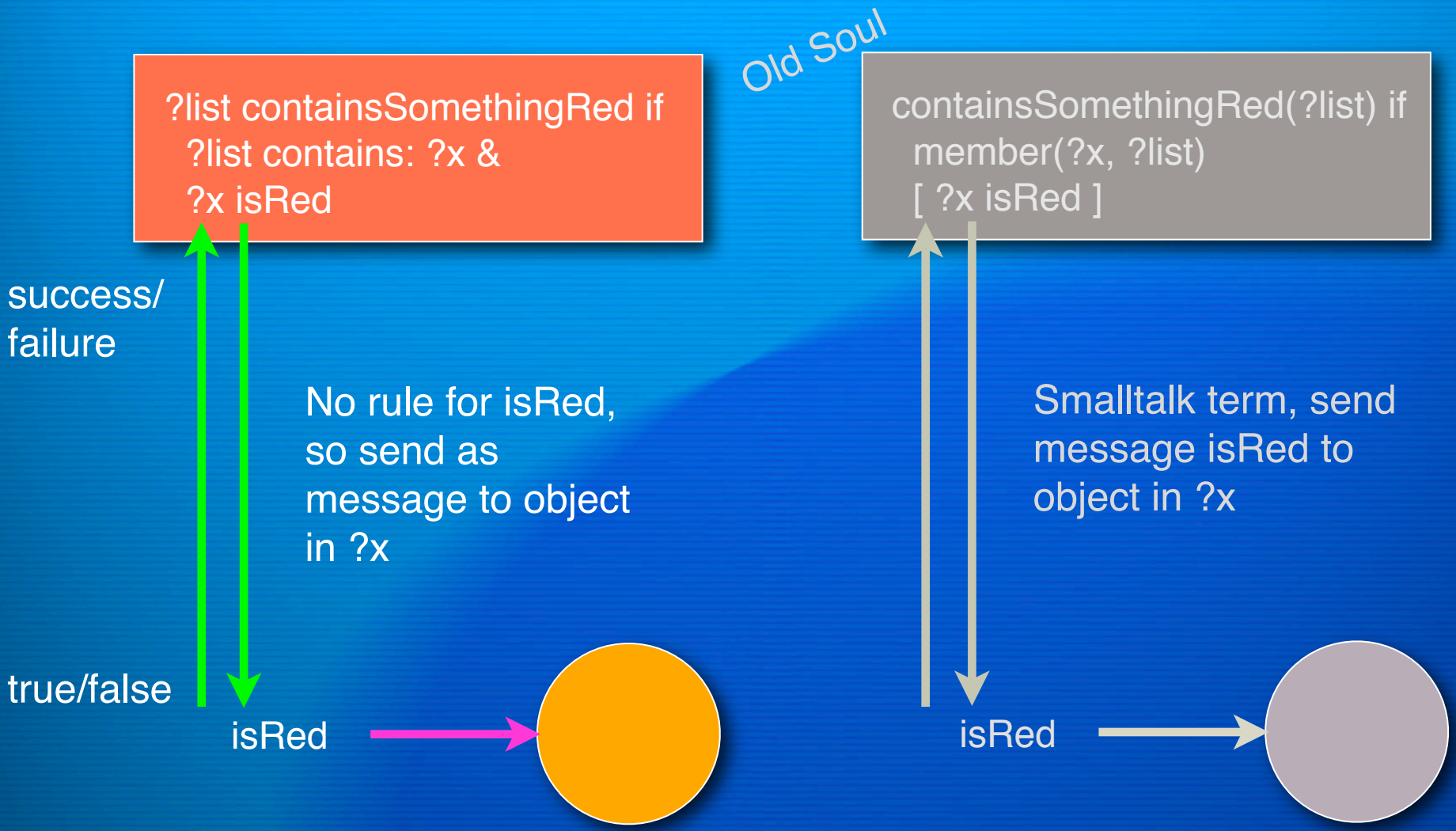
No rule for isRed, so send as message to object in ?x



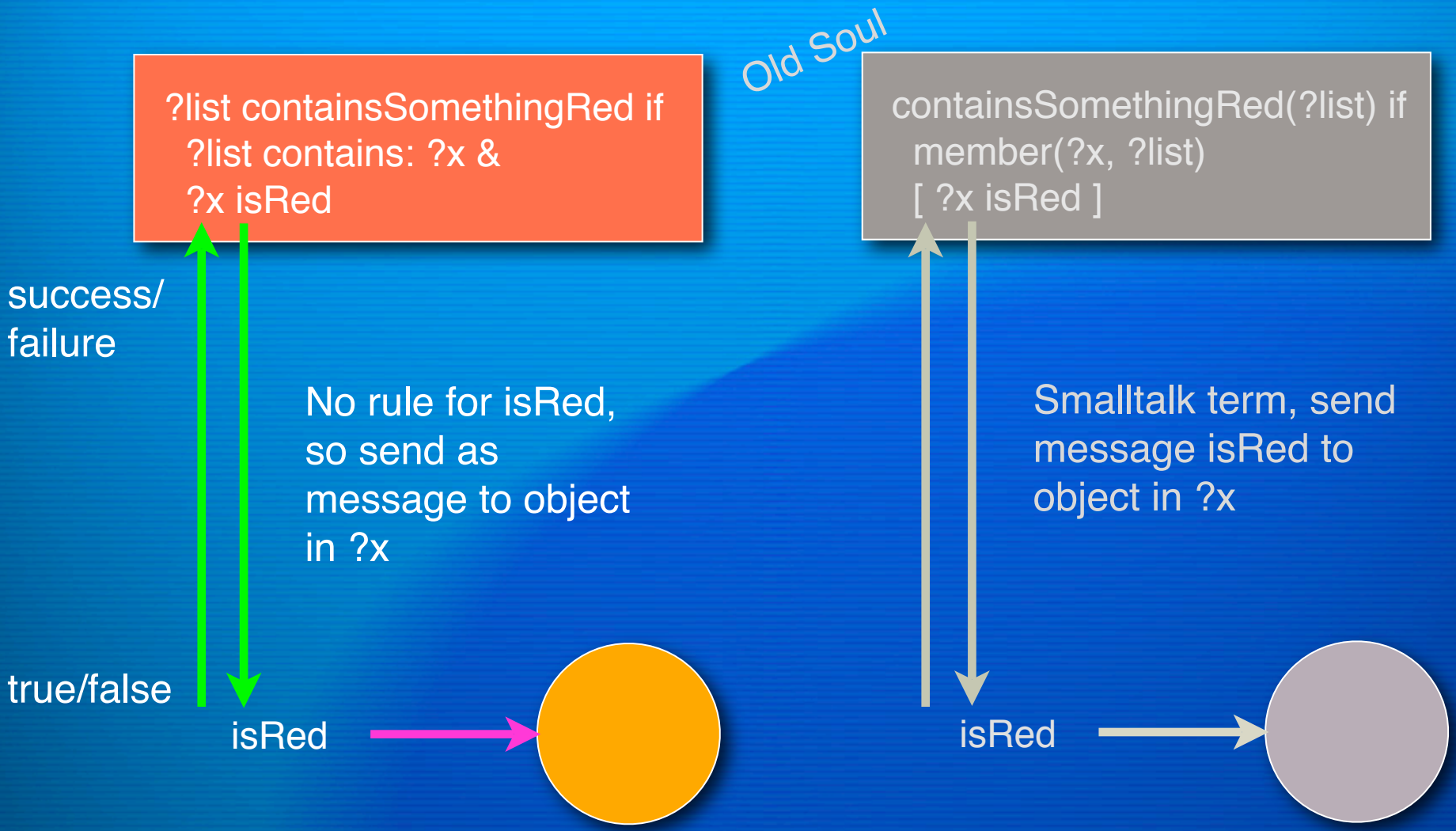
New Interpretation Process



New Interpretation Process



New Interpretation Process



Note: defining a rule for isRed would override message sending