Position Paper: Interconnection Reflection

H. Justin Coven
Department of Computer Science
Bellarmine College
Louisville, KY 40205
hjcove01@ulkyvx.louisville.edu

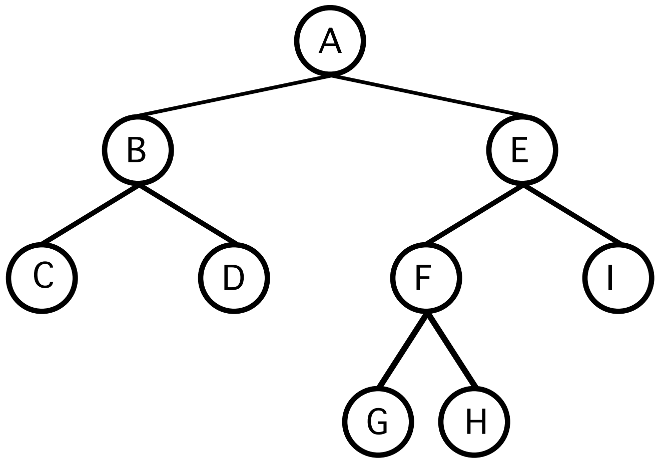
Background

My work has been in general reflection not reflection specially for object-oriented programming languages. I have looked at reflection across numerous domains including object-oriented, logic and functional domains. One of my major concerns has been the ability of a reflective system to be able to reflect between numerous domains. Because of this I have looked for a common denominator as a base representation for these diverse systems to work with. Since all of these systems need to be implemented, I have chosen as a base denominator an abstract representation of implementation - the operational semantics domain.

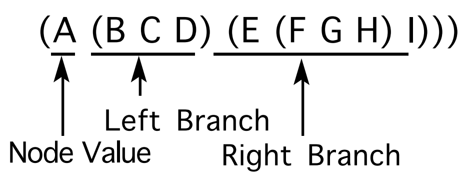
In my dissertation[1] I explored what are the basic components of any system in the operational domain: data, functions, control, interconnections, interpreter and learning. Data would include symbols, numbers, characters, lists, as well as other basic structures and types. Functions include procedures, functions, predicates, methods and like minded constructs. Control contains the information of where execution has progressed to so far, what are the temporary computations - this includes all local variable values as well as position indication. For the most common computational model - procedural - this would be contained in a control stack. Interconnections are the links between pieces of information. Any form of scoping or inheritance indicate specific rules for how components (data, functions, or control) are connected and how they can be traversed are such links. The interpreter contains tools for accepting specific forms of syntax, the general execution system, and specific sets of instructions not related to any of the above four components. Learning includes all tools that help a system modify itself no matter which component the system is modifying - data, functions, control, interconnections, interpreter, or even learning. In the terms that appear in the flyer for this workshop my classification of reflection is inward=downward and not outward. Here I equate operational definition (inward) and implementation (downward) and do not consider reflection into the operating system, database domains, or other domains (outward). However, in my work on control reflection I have created language constructs that can take on some of the "process" prioritization tasks that are typically handled by the operating system (control carousels in [1]).

My focus of research for the last few years has be in reflection upon control[2]. What follows is a brief description of some of my previous work. I give three of the briefest examples of what control reflection can do. In the first example we meet control queues which are a variant to the standard control stack implementation of procedures (functions with no return values). With a control stack the control frame that called the current control frame is returned to, thus control frames are stacked up. With control queues, the front frame in the control queue is in control, new control frames are placed on the back and wait their turn to execute. Control queues can be used to use recursive function calling to do breadth-first traversal as opposed to the standard depth-first search of control stack recursive function calling. Additional control mechanisms have been devised to do best-first traversal.

Let's look at a simple example of breadth-first traversal. in the below figure we give a list representation of a tree.



A. A tree to be searched Breadth-First



B. A list representation of the same tree.

We define function "q-ord" below with the built-in function "defq" instead of "defun" otherwise the function is identical to how a depth-first control stack recursive function would appear.

```
qlisp> (defq q-ord (tree)
  (if (atom tree) (prin1 tree)
    (progn
      (prin1 (first tree))
      (q-ord (second tree))
      (q-ord (third tree)))))
Q-ORD                    -return value
qlisp> (q-ord (quote (a (b c d) (e (f g h) i))))
ABECDFIGH               -printed values
NIL                      -return value
```

The alternative depth-first function's output would be:

```
ABCDEFGHI               -printed values
I                        -return value
```

With control queue's we modified the control structure mechanism. With our next example we demonstrate a tool that explicitly accesses any part of the standard control stack. The built-in function "grab" will return the control frame indicated by a position argument given to it. The position is the location below the current control frame within the control stack. Note that in the below example the argument is 0 thus the current control frame is returned. The below example is a recursive lambda function. This is significantly more understandable and simpler than the typical Y-operator solution to create a recursive lambda.

```
qlisp>  (funcall (function (lambda (n)
                    (if (= n 0)
                        1
                        (* n (funcall (code (grab 0)) (- n 1))) )))
         7)
5040                     -return value
```

The function "code" is a built-in that returns the code component (or function definition) of the control frame given to it as an argument. In our implementation it is equivalent to the first or car functions. Recursive lambda is useful if you want a program to modify or create functions - any truly intelligent program needs to create it's own reasoning thus the necessity for this capability. We also have created a tool that can explicitly manipulate the control stack. Thus information that has been typically been difficult if not impossible to access and manipulate explicitly (control knowledge) is now accessible and manipulatable.

Our last example has the greatest amount of reflection, it allows a program to modify its control mechanism as the program is running. The control mechanism is converted from a control stack to a control queue when the built-in function "converttoq" is executed.

```
qlisp> (defun print12 (list)
  (if (null list)
     (converttoq print12)
     (progn (prin1 (first (first list)))
          (print12 (rest list))
          (prin1 (second (first list))))))
PRINT12                  -return value
qlisp> (print12 (quote ((a 1) (b 2) (c 3))))
ABC123                   -printed values
NIL                      -return value
```

Here we have used the converttoq to traverse a list from front to back twice in a row without having to have two functions of two calls to a function. The list is traversed first in a depth-first fashion. Then the control mechanism is turned into a control queue and the first function call is put in charge, as opposed to the most recent frame for control stack. The first function call had the first entry in the list as the part of the argument it worked on..

Position

The current work in object-oriented reflection appears to me to be focused on reflection on interconnections. Most work has focused upon how to manipulate the search for variables and methods along interconnections, as well as manipulating the tasks done while that search is going on. Tasks such as message interception, method identification, prioritization, method combination, method execution, and return[3,4,5]. All of these are forms of interconnection link traversal.

My own position for the advancement of reflection in object-oriented programming is that we need to extend the Object-Oriented model so that we can reflect on interconnections in general. This would include syntax general enough to accept models such as Frames and Semantic Networks. Both systems have interconnections and link traversal algorithms associated with those interconnections. Our Object-Oriented model needs to subsume both. Currently our model effectively subsumes the Frame model, but because of the freedom of the Semantic Network model in creating any type of and number of links with self defined algorithms for traversal the Semantic Network model is not subsumed. In Semantic Networks some links have specific properties such as link traversal algorithms (e.g. is-a, part-of), while others just have a link name and use a general traversal algorithm that only goes along sets of links only of matching names. Thus in order to subsume the Semantic Network model we need to be able to allow named links (not just a single kind of standard inheritance link), as well as the ability to fully define the traversal algorithm for different kinds of links, for specific links, and for the interaction between different links.

Stating my position in another way, I am answering a question asked at an earlier workshop[6].

> "Is reflection nothing but object-oriented programming done right? or was object-oriented programming reflection all along?"

I answer by stating what I believe is the relation between object-oriented programming and inward=downward reflection. Object-oriented programming is really an implementation mechanism that takes a specific language syntax and has a specific interconnection traversal protocol. Currently there has been much research into different kinds of interconnection protocols[7-11], all of them proving useful in different instances. What has become necessary are mechanisms to choose between these protocols and potentially create new protocols, and this is what inward=downward "reflection" on object-oriented systems is. We need to open object-oriented reflection open wider. Currently the syntax that is accepted by object-oriented systems is not conducive to develop any number of different kinds of interconnections. We need to open reflective object-oriented programming to things such as Semantic Networks where you can have any number of any kinds of links-interconnections.

Of course in allowing any type of link we must take care. In the history of developing Knowledge Representation systems Semantic Networks proved to have too unstructured types of links that ran into computational explosion problems, thus the Frame model was developed and object-oriented systems took much from this development. Like Frames, object-oriented systems have well defined inheritance links that clearly define and limit the amount of traversal. However with the explosion of research into different kinds of interconnections, we are again running into problems of ambiguousness of how to traverse multiple links (multiple-inheritance).

As an example of a potential syntax that would accept not only multiple links but multiple definitions of links we might have:

(defclass-object name SuperClassLinkName SuperClassName SuperClassLinkName2 SuperClassName2 ... PrecedenceRulesForLinks "variables" "methods")

Where SuperClassLinkName is a link name that identifies the kind of link and SuperClassName indicates the superclass the name is linked to. A set of rules on how to combine different kinds of links may also be needed, which is what PrecedenceRulesForLinks is for.

As an example of the depth of problem we need to handle, consider all of the different kinds of links needed in the model of a boy holding a door handle and then walking. Does the door handle go with him? By default-common sense it should not. If there is an exception that it is not attached then it should go with him. If there is an exception that he doesn't let go then he does not move while walking. While by default it he is walking then he should be moving. Which link - walking or holding the door handle breaks? If there is an exception that the door is not attached to a wall then the door knob goes with him as well as the door. How do we follow all the proper connections, as well as modify the model while following the links to insure that the door and door knob move to the same location as the boy. How do we define a type of link that will modify position of nodes as it traverses down links to determine which connection is stronger? This type of problem necessitates the creation of many different types of links and associated traversal algorithms.

1 Coven, H. Justin, *A Descriptive-Operational Semantics for Prescribing Programming Languages with "Reflective" Capabilities*, 1991, Ph.D. dissertation, Arizona State University.

2 Coven, H. Justin, "Control Tools and their Implementation", Work in Progress.

3 Foote, B. and Johnson, R. E. Reflective Facilities in Smalltalk-80. *ACM SIGPLAN Notices* 24(10): PP327-335; October, 1989.

4 Cointe, P. Metaclasses are First Class: the ObjVlisp Model. *ACM SIGPLAN Notices* 22(12): PP156-167; December, 1987.

5 Kiczales, G., Lamping, J., Bawden, A., des Rivieres, J., Dixon, M. and Smith, B. Object Oriented Meta Level Architectures. Unpublished, all authors at Xerox PARC except A. Bawden who is at MIT.

6 Ibrahim, M. H., "Workshop: Reflection and Metalevel Architectures in Object-Oriented Programming", *SIGPLAN Notices*, Special Issue, OOPSLA/ECOOP '90 Addendum to the Proceedings, 21-25 October 1990.

7 America, P. and van der Linden, F. A Parallel Object-Oriented language. *ACM SIGPLAN Notices* 25(10): 161-168; October, 1990.

8 Danforth, S. and Tomlinson, C. Type Theories and Object-Oriented Programming. *Computing Surveys* 20(1): 29-72; March, 1988.

9 Helm, R., Holland, I. M., and Gangopadhyay, D. Contracts: Specifying Behavioral compositions in Object-Oriented Systems. *ACM SIGPLAN Notices* 25(10): 169-180; October, 1990.

10 Lalonde, W. R. and Pugh, J. R. *Inside Smalltalk: Volume II.* Englewood Cliffs, NJ: Prentice Hall; 1991.

11 Pedersen, C. H. Extending Ordinary Inheritance Schemes to Include Generalization. *ACM SIGPLAN Notices* 24(10): 407-417; October, 1989.