Design Issues in a Minimal Language to Support Lisp-based, Object-based, and Rule-based Programming

S. Fickas, E. Doerry,
D. Meyer and P. Miller

CIS-TR-88-16
December 30, 1988

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Design Issues in a Minimal Language to Support Lisp-based, Object-based, and Rule-based Programming

Stephen Fickas, Eckehard Doerry, David Meyer, Peter Miller

Computer Science Department
University of Oregon
Eugene, OR. 97403

Abstract

In this paper we argue for a minimalist view of language design for Expert System environments. In support of our arguments we present MIN, a minimal language which extends the less-is-better philosophy of Scheme to include both object-based and rule-based components. We discuss the strengths and weaknesses of MIN as an embodiment of the minimalist philosophy, and point to other work supporting this view of language design.

1. Introduction

Our interest is in the integration of lisp-based, object-based, and rule-based language components into a single framework. In this paper, we describe a language that attempts to cleanly and simply represent all three.

There are three fundamental design principles that guide our work:

Principle 1: *Less is better*. Our goal is to provide the minimal number of primitive language components to support lisp-based, object-based, and rule-based programming. In many ways our approach is the antipode of languages like CLOS [Keene, 1988] and other Lisp based AI shells, which attempt to provide a broad and complex set of functions to their users. Our work tests the opposite pole: is there a simple, minimal language that can act as the foundation for more complex extensions.

We have chosen Scheme as our lisp-based component because of its balance of power and simplicity. This paper, in essence, discusses our effort to create a similar balance in the object and rule components of our language.

Principle 2: *Integration versus aggregation*. Given Scheme as a foundation, our goal becomes the integration of object-oriented programming (OOP) and rule-based programming (RBP) into Scheme. In general, our goal is to avoid extensions to Scheme whenever possible, and when forced to extend it, find sound justification for the extension.

Principle 3: *Efficiency is a key concern*. Our goal is to build an abstract machine model of Scheme, RBP, and OOP that will act as the basis for efficient compilation and execution. Such models exist for Scheme, object-based languages, and rule-based languages separately; our work attempts to integrate the three.

In the remainder of the paper, we will describe our language, MIN, and in particular, the design of its OOP and RBP components in relation to principles 1 and 2. A discussion of our efforts to bring about an efficient implementation of MIN (principle 3) is taken up in a separate paper [Fickas *et al*, 1988].

2. The Object Oriented Programming component

Following the principles above, the integration of OOP into Scheme has lead us to consider the following implications:

- Objects should be first-class citizens.
- Slots should uniformly represent state and functionality, and of course, be first-class citizens.
- Lexical scoping within the object system should be preserved whenever possible.

In defining MIN, we have chosen a delegation model of OOP [Lieberman, 1986] over a class-instance model. The equivalence of the two models has been argued both informally [Lieberman, 1986] and formally [Stein, 1987]. Our choice was motivated by our desire for simplicity.

We have further simplified the delegation model as described in [Lieberman, 1986] by constructing objects solely out of two types of entities: *generic slots* that can represent both state and functionality, and *parent objects*[1]. In particular, there is no distinct representation of a method.

[1]Both the term *parent objects* here, and *inheritance* later, are misleading: the delegation model does not promote the class-instance view associated with either term. Lieberman would replace *parent* with *prototype*; we prefer *shared objects*. However, we will stick with parent and inheritance because of their common usage.

[2]Notation: normal quote is represented by ' ; quasi-quote or backquote is represented by `. A comma represents an eval-and-insert operation within a backquoted list.

As an example, to define an object in MIN that represents a geometric point in space, we use the following[2]:

```
(define slot1  (create-slot '(  (name x-coord)
                                (value ,init-val-for-x)
                                (dtype real))))

(define slot2  (create-slot '(  (name y-coord)
                                (value ,init-val-for-y)
                                (dtype real))))

(define p1  (create-object '(,point-operations)  '(,slot1 ,slot2)))
```

The first argument to create-object is a list of objects to use as parents, in this case an object point-operations, created elsewhere, that defines a set of operations on points. In general, inheritance is based on pre-order traversal of the parent list.

Note that the variable point-operations is bound to another object - it is not simply a symbol that somehow internally represents another object (c.f., Flavors [Moon, 1986]). In other words, objects (and slots) are first-class in MIN.

The second argument is a list of slots. A slot is structured as a list of *facets*. Two facets, name and value, are predefined; every slot must contain at least these two facets. The user may define further facets to fit the application (e.g., dtype above).

Slots can be added after object creation using the add-slot! function. Conversely, slots can be deleted from an object using delete-slot!.

Slot retrieval is accomplished using the primitive function get-slot, defined for all objects. The arguments of the get-slot function are an object and a list of slot descriptors that describe, using facets, the slot or set of slots that are of interest. As an example:

```
(get-slot obj1 '((name slot1) dtype))
```

will return the slot which has a name facet whose value is slot1 and a dtype facet whose value is anything. Multiple arguments are treated as a conjunctive description.

As another example,

```
(get-slot obj1  '((value ,pattern-val)))
```

will return all slots that have a value facet equal to the value of pattern-val.

A null list of facet descriptors is read as unrestricted lookup, and will cause a list of all slots of an object, including ones inherited, to be returned.

```
(get-slot obj1  '())
==>  <list of all slots accessible from obj1>
```

A slot itself, can be accessed through the functions add-facet!, delete-facet!, get-facet, and set-facet!.

We have now discussed one half of an object, its slots. That leaves us with the second half, its list of parents. One can access the parents of an object as follows:

```
(get-parents obj1)
```

This returns a list of objects that are the immediate parents of obj1. One can also reset the parent list, destructively altering the inheritance hierarchy:

```
(set-parents! obj1 `(,obj2 ,obj3))
```

Finally, there is a message defined for searching just the slots attached directly to an object, in essence, ignoring inheritance.

```
(get-local-slot obj1 `((value ,pattern-val)))
```

In conjunction with the get-parents function, we can use get-local-slot, among other things, to experiment with other inheritance schemes than the standard pre-order traversal built into get-slot.

This completes the set of primitive functions defined for objects and slots in MIN.

2.1 Methods

In some sense this section could be omitted: there are no intrinsic method types in our language. However, it is clear that some type of procedural mechanism must be associated with an object. We use lexical closures, in conjunction with our generic slots, to implement this OOP component in MIN.

In particular, a "method" can be defined by storing a Scheme lexical closure within a slot. As an example,

```
(define method1 (lambda (x) (+ x 1)))

(add-slot! obj1 (create-slot `((name meth1) (value ,method1))))
```

will add a new slot meth1 to obj1 whose value is the procedure which adds 1 to its single argument. To use this method/procedure, we do the following:

```
((cadr (get-facet (car (get-slot obj1 '((name meth1)))) 'value)) 6)
==> 7
```

To simplify subsequent examples, we will define the following functions to carry out getting a slot value and applying a method.

```
(define get-slot-value (lambda (obj slot-name)
    (let* ( (slots (get-slot obj `((name ,slot-name))))     ;list of slots
            (slot (car slots))                              ;first slot
            (facet (get-facet slot 'value)))                ;facet pair
      (cadr facet))))                                       ;value

(define send (lambda (obj slot-name . rest)
      (apply (get-slot-value obj slot-name) rest)))
```

It is important to note that a procedure placed in a slot carries normal lexical closure. Thus,

```
(define obj1 (create-object (...) ...))

(let* (   (secret-password 'Portland-in-5)
          (method2 (lambda (p x) (if (eq? p secret-password)
                                     (+ x 1)
                                     0))))
    (add-slot! obj1 (create-slot `((name meth2) (value ,method2)))))
```

will place the lexical closure of method2, including the binding of secret-password, into the value facet of the meth2 slot of obj1, *no matter where obj1 was initially defined.*

2.2 Self

We have yet to provide a way for a procedure in one slot to reference other slots. For instance, suppose we are given the following objects:

```
(define obj1 (create-object '() '())

(define obj2 (create-object `(,obj1) '((x 1) (y 2))))

(define obj3 (create-object `(,obj1) '((x 5) (y 6))))
```

and now want to add a method to obj1 that will return the sum of a pair of x and y slots, and multiply the result by some constant:

```
(let* (  (constant 5)
         (add-mult  <method>))
   (add-slot! obj1 (create-slot '((name addxy) (value  ;add-mult))))

(send obj2 addxy  <method args>)
==>  15

(send obj3 addxy  <method args>)
==>  55
```

What should be the value of <method> above? Our first approach at defining <method> might be as follows:

```
(lambda () (* (+ x y) constant))
```

where <method args> in the send call would be unneeded. This captures the notion that constant is lexically scoped, but the scope of x and y remain problematic.. In particular, we want x and y to be looked up dynamically, depending on which object is invoking the method.

We could force x and y (and all other slot references) to come in as arguments (see, for instance, [Adams&Rees, 1988]). Hence, <method> would be defined as

```
(lambda (x y) (* (+ x y) constant)
```

and the actual method invocation would be

```
(send obj2 addxy  (get-slot-value obj2 'x) (get-slot-value obj2 'y))
==>  15

(send obj3 addxy  (get-slot-value obj3 'x) (get-slot-value obj3 'y))
==>  55
```

This approach is cumbersome in cases where there are a large number of slot references within the body. It also forces an eager and potentially inefficient lookup policy.

What we propose is to allow a procedure body to reference a variable, self, freely, i.e., as if it was within the lexical scope of the procedure. The system will bind self to the correct object, and place self within the lexical scope of the method *at run time*. With this approach, <method> becomes

```
(lambda (x y) (* (+  (get-slot-value self 'x)
                     (get-slot-value self 'y)) constant))
```

and method invocation becomes

```
(send obj2 addxy)
==> 15

(send obj3 addxy)
==> 55
```

In the first call on send above, self is bound to obj2; in the second it is bound to obj3.

To implement this use of self, we provide a primitive MIN function, extend-closure, that takes as arguments an existing procedure (closure) P and a list of variable-value binding pairs B. It returns a new procedure P' that consists of the environment of P extended to include B as an enclosing environment of P. This can be used to implement the following alternative form of get-slot-value:

```
(define get-slot-value (lambda (obj slot-name)
        (let* (   (slots (get-slot obj '((name ,slot-name))))   ;list of slots
                  (slot (car slots))                             ;first slot
                  (facet (get-facet slot 'value))                ;facet pair
                  (facet-value (cadr facet)))                    ;value
           (if (procedure? facet-value)
               (extend-closure facet-value '((self ,obj)))
               facet-value))))
```

Note that the extend-closure function can be used to define other types of "free dynamic variables" as if they were lexical. For example, with the right call to get-parents, we can define a new form of get-slot-value that includes super:

```
(define get-slot-value (lambda (obj slot-name)
        . . .
        (if (procedure? facet-value)
             (extend-closure facet-value '((self ,obj) (super ,parent)))
             facet-value))))
```

A more detailed discussion of the issues of mixing the lexical scoping of Scheme with the dynamic scoping of OOPS is presented in [Fickas et al, 1988].

3. The Rule Based Programming component

As with integrating OOP into Scheme, our goal is to avoid extensions specific to RBP alone. This has lead to the following implications:

- Rules should be objects (that match on other objects).
- Databases should be objects (containing other objects).
- Rule matching must be integrated with our delegation model of objects.
- The components of a rule-based system (rbs) should be objects. This includes databases *and* procedural components, e.g., matchers, conflict resolvers, rule invokers.

- A rbs, itself, should be an object.

The representation of a rule-based system in MIN is the *rbs* object. The rbs object consists of the following slots:

rbase: value is a rule database.

fbase: value is a fact database.

matcher: value is a matcher-object.

cs: value is a rule-instantiation database.

conflict-resolver: value is a conflict-resolver-object.

es: value is a rule-instantiation database.

engine: value is an engine-object.

All databases (rbase, fbase, cs, es) share, through delegation, the same general object, *object-collection*. Procedural objects (matcher, conflict-resolver, engine) share the same general object, *rbs-method*.

The engine-object specifies the overall control cycle of the rbs. The default engine in MIN models the forward chaining control cycle such as found in OPS-5 and YAPS:

1. A set of rules is gathered from a rule-base.
2. A set of facts is gathered from a fact-base.
3. A matcher is called to generate the conflict set.
4. A conflict resolver is called to filter the conflict set, producing the execution set.
5. The execution set is sent to a rule invoker, causing rule actions to be carried out.
6. The cycle repeats.

Any rbs slot can be changed at run-time. Typically this involves changing the rule-base or the fact-base. However, it is also possible to change the other slots, including the matcher or even the engine.

3.1 Rules

A rule consists of a condition part and an action part. The condition part, in turn, consists of a set of object pattern-matching clauses (declarative) and a set of matching filters (procedural). The action part of a rule consists of one or more evaluatable MIN expressions.

The primitive function `create-rule` is used to define a rule:

```
(define rule1 (create-rule ( <clause1> ... <clausen> )
                           ( <filter1> ... <filterm> )
                           ( <action1> ... <actionk> )))
```

Each `<clause>` has the following form:

```
(object-label (slot-name slot-value) ... (slot-name slot-value))
```

The object-label is optional; if present, it is bound to whatever object matches the clause. The slot-name and slot-value can be either a literal or a pattern variable. Thus, object matching is fully associative on the two facets name and value.

Rule clauses match on objects in object-collections. Object-collections are objects that can hold any MIN object. Thus, rules can match on application objects and rbs objects as well, e.g., other rules, rule instantiations, rbs-methods, other object-collections, or even other rbs objects.

The actions of a rule are invoked within the lexical scope of the rule. Thus, the action (foo -x -y) in rule1 would look for -x and -y in the clauses of rule1. Of course, references to -x or -y within the body of foo will be resolved in the environment where foo is defined, i.e., foo itself retains normal lexical scoping.

Rules, being first class objects, can be passed as values, and shared among various rule-bases:

```
(send rbase1 'add-object rule1)
(send rbase2 'add-object rule1)
```

3.2 Object collections

The MIN object object-collection is a generic representation for a heterogeneous collection of MIN objects. Individual object elements within the collection are stored as slots with a facet of (stype object-element). The object collection has three major benefits and at least one major drawback. The benefits first.

Benefit: *Gathering the objects in an object collection is trivial:*

```
(get-slot db1 '((ftype object-element)))
==> <list of all objects visible from db1>
```

Benefit: *Object lookup can be integrated with delegation:*

Thus, in the previous call to get-slot, we will return not only the objects in the object-collection db1, but also any objects seen by db1 through delegation. This allows us to share databases in arbitrarily complex ways. For instance, we might set up a global database as a type of blackboard shared by all rbs objects, and at the same time, allow each individual rbs to define its own local database. It we link the local databases to the global one, through delegation, then we have a public-private rbs architecture. A wide variety of other architectures are possible.

Benefit: *Object collections can be shared, in interesting ways, among rbs.*

As discussed above, the same object collection can be shared among various rbs. Thus, the same collection of rules or facts can be shared, either by direct installation or through delegation, by more than one rbs.

Typically such sharing is uniform: two rbs use an object collection as a joint fact base or a joint rule base. However, this does not have to be the case. For instance, the rule-base of rbs1 can be used as the fact base of rbs2, where rbs2 may contain its own rule base that attempts to add, delete, or refine the rule base of rbs1 (as noted, object-collection is predefined by the system):

```
(define dbshared (create-object '(,object-collection) ... ))

(define rbs1 (create-object '(,rbs) '(   (rbase ,dbshared)
                                         (fbase ,facts) ... )))

(define rbs2 (create-object '(,rbs) '(   (rbase ,refinement-rules)
                                         (fbase ,dbshared) ... )))
```

In another example, the same object collection may be used by rbs1 as a conflict set, and by rbs2 as a fact base. Here, we would expect rbs2 to be the conflict resolver of rbs1, i.e., rbs2 would be installed in the conflict-resolver slot of rbs1 (an rbs object can act as an rbs-method as well):

```
(define dbshared (create-object '(,object-collection) ... ))

(define rbs2 (create-object '(,rbs) '(   (rbase ,meta-rules)
                                         (fbase ,dbshared) ... )))
(define rbs1 (create-object '(,rbs) '(   (rbase ,rules)
                                         (fbase ,facts)
                                         (cs ,dbshared)
                                         (conflict-resolver ,rbs2)  ... )))
```

Drawback: *A simple RETE style matcher will no longer function in the face of our more general view of RBP in* MIN.

The original RETE algorithm was devised for a single rbs working on a single fact base with a single rule base [Forgy, 1982]. In MIN, we have generalized this to a rule-based system where 1) more than one rbs may be extant at any one time, 2) object collections can be shared among rbs, and 3) delegation can be used in object lookup during matching.

If we wish to retain the benefits of doing incremental matching in a RETE style, then we must devise a more general model of incremental matching. We have done so - the details are discussed in [Fickas *et al*, 1988]. Briefly, we allow an rbs that uses an object collection to register with that collection. This registration includes a time stamp indicating the last time at which the rbs was updated. The object collection will then keep track of changes since that time, and provide the changes on demand when the rbs is ready. While this two sentence explanation leaves out many details, two aspects of our matching algorithm warrant emphasis: 1) it is lazy in that changes are only supplied to an individual rbs on demand, 2) it avoids direct pointers from object collections to the rbs that employ them. Both of these appear to be necessary features in several distributed models of MIN that we are exploring.

4. Related work

There are two projects that have explored the integration of OOP with Scheme: [Lang&Pearlmutter, 1988], [Adams&Rees, 1988]. Both have made strong contributions to the area. One might view MIN as an attempt to bring together the best of both of these projects -- the uniform view of objects and methods as first-class in Oaklisp, and the delegation based model of OOP in T -- while following the minimalist philosophy laid out in section 1.

Also related to the OOP component of MIN, the Self language [Ungar&Smith, 1987] is based on the integration of a delegation model within a Smalltalk like language. As with our language, Self makes no distinction between ''state'' slots and ''behavior'' slots. One of the main arguments put forth in the Self project is that of unification: lexical scoping and closures can be unified with delegation-based inheritance. While this is a seductive goal -- for instance, aren't the environment diagrams of Scheme [Abelson&Sussman, 1985] just inheritance graphs -- we do not believe that Self has demonstrated this unification, nor are we convinced that it is desirable. In particular, our view is that OOP is inherently dynamic in nature. While it may be possible to mutate an OOP system to model Scheme, we expect that the resulting system would no longer be object-oriented in nature. In short, we believe that there is a place for both OOP style and Scheme style semantics in a single language.

Regarding the RBP component of MIN, the foundations of our approach were first presented in [Fickas, 1985]. Later work attempted, on a more limited basis, to integrate a Smalltalk view of OOP with RBP [Laursen&Atkinson, 1987].

5. Conclusions and future work

What we have presented in this paper, in reality, is a philosophy of language design for AI environments (among others), and a first prototype language built under that philosophy. We can make no claims that the prototype is the simplest or most minimal: the former would require quite subjective criteria, and the latter arguments from abstract language theory that we find of little practical interest at this point in our project. What the prototype can demonstrate is the outcome of a diligent effort to follow the *less is better* philosophy. As with Scheme, our design of MIN has been driven by a desire of elegance through simplicity. While it is clear to us that we have not fully achieved our goals in MIN -- debate within our group about almost every MIN component continues unabated -- we remain convinced that they are worthy goals to pursue.

Conversely, we note that work on complex languages such as CLOS plays a valuable role: they explore the type of functionality we must eventually provide to build practical systems. In our project in particular, CLOS is one of several target systems that we feel we must be able to construct on top of MIN to be successful.

The MIN system -- the extended Scheme abstract machine, the OOP component, the RBP component, and a copy-based garbage collector -- is implemented in C and MIN (taking a minimum of 512K bytes), and has been ported to UNIX based, Mac, and PC hardware.

The majority of our current effort goes towards refining the components of the MIN system, and studying means for their efficient implementation. We also have started work on the addition of a logic-based component to MIN, modeled as just another rbs object [Doerry, 1988]. The interesting aspect of this work is the concept of multiple Prolog-style systems 1) in existence simultaneously, and 2) sharing databases. Each of these ideas has seemed to flow naturally and simply from the root MIN implementation.

Acknowledgments

Will Clinger and Mark Vanderwettering provided useful insight into both the Scheme and OOP components of our language. We acknowledge the National Science Foundation under grant DCR8312578, Tektronix, Sun Microsystems, NEC, and Keicho, Ltd. for their support of this work.

References

[Abelson&Sussman, 1985] Abelson, H., Sussman, G., *The structure and interpretation of computer programs*, MIT Press, 1985

[Adams&Rees, 1988], Adams, N., Rees, J., Object-oriented programming in Scheme, *Conference on Lisp and Functional Programming*, Salt Lake City, 1988

[Doerry, 1988] Doerry, E., Caching in logic-based systems, TR 88-24, Computer Science Department, University of Oregon, 1988

[Fickas, 1985] Fickas, S., Design issues in a rule based system, In *ACM Symposium on Programming Languages and Programming Environments*, Seattle, 1985 (revised version to appear in *Journal of Systems and Software*, 1989)

[Fickas *et al*, 1988] Fickas, S., Doerry, E., Miller, P. Nitzberg, W., The MIN project: design issues in a minimal language to support lisp-based, object-based, and rule-based programming, TR 88-23, Computer Science Department, University of Oregon, 1988

[Forgy, 1982] Forgy, C., RETE: a fast algorithm for the many-problem/many-object pattern matching problem, *Artificial Intelligence*, No. 19, 1982

[Keene, 1988] Keene, S., *Object-oriented programming in Common Lisp: a programmer's guide to CLOS*, Addison Wesley, 1988

[Lang&Pearlmutter, 1988] Lang, K., Pearlmutter, B., Oaklisp: an object-oriented dialect of Scheme, *Lisp and Symbolic Computation*, Volume 1, Number 1, Pages 39-51, 1988

[Laursen&Atkinson, 1987] Laursen, J., Atkinson, R., Opus: a Smalltalk production system, *Proceedings of OOPSLA '87*, Orlando, Florida, 1987

[Lieberman, 1986] Lieberman, H., Using prototypical objects to implement shared behavior in object oriented systems, *Proceedings of OOPSLA '86*, Portland, Oregon, 1986

[Moon, 1986] Moon, D., Object-oriented programming with Flavors, *Proceedings of OOPSLA '86*, Portland, Oregon, 1986

[Stein, 1987] Stein, L., Delegation is inheritance, *Proceedings of OOPSLA '87*, Orlando, Florida, 1987

[Ungar&Smith, 1987] Ungar, D., Smith, R., Self: the power of simplicity, *Proceedings of OOPSLA '87*, Orlando, Florida, 1987