TECHNICAL REPORT NO. 276

Embedding the Self Language in Scheme

by

Julia L. Lawall and Daniel P. Friedman

May 1989

COMPUTER SCIENCE DEPARTMENT

INDIANA UNIVERSITY

Bloomington, Indiana 47405–4101

Embedding the Self Language in Scheme*

Julia L. Lawall†
Computer Science Department
Indiana University
Bloomington, IN 47405, USA
jll@iuvax.cs.indiana.edu

Daniel P. Friedman‡
Computer Science Department
Indiana University
Bloomington, IN 47405, USA
dfried@iuvax.cs.indiana.edu

Abstract

Two common approaches to language implementation are interpretation and compilation. In this paper we describe a third approach, the process of embedding a language within another. This approach combines speed of implementation with efficiency of the resulting language. A language defined in this manner is also more flexible than its compiled or interpreted counterparts. We present an implementation of the object-oriented programming language Self as an example of this technique.

1 Introduction

Two common approaches to language implementation are interpretation and compilation. These differ in the ease of implementation and the speed of the defined language [Reynolds72], but they share the property that the defining language is closed off from the host language. It may, however, be desirable for the new language to be able to interact with the language in which it is implemented. The new language may not be suitable for all applications. If it can interact with the defining language and other languages are also embedded in that language, then the new language can easily interact with those languages as well. This capability adds to the flexibility of the system.

One language in particular with which the defined language can interact is the defining language itself. The new language can inherit such features as the definition and application of primitive functions. Thus the implementation of the new language is simpler. The implementor can concentrate only on those aspects that differ from the host language. Of course the programmer then has to avoid those features of the host that are intended to be replaced by the embedded language.

To illustrate the ease of the embedding approach to language implementation we have chosen to embed the language Self in Scheme [Rees86]. Self [Ungar87] is an object-oriented programming language that has no classes, only instances. Self has a flexible system of inheritance and a uniform way of accessing variables and methods, comparable to that of Eiffel [Meyer88]. Using our implementation we can explore the interactions between Self and other languages that can also be embedded in Scheme, such as Prolog [Felleisen85]. Our approach may be distinguished from that of Adams and Rees [Adams 88] in that we embed an existing language into Scheme, whereas they add object-oriented features to Scheme itself.

The next section describes the Self language in greater detail. Embedding Self in Scheme is made practical by Scheme's mechanism for defining new syntax, `syntax`. This mechanism was developed by Eugene Kohlbecker

*To appear in the *BIGRE Bulletin*'s special issue on "Putting the Scheme Language to work"

†Supported by a NSF Fellowship.

‡Supported by the National Science Foundation under Grants Numbered CCR 87-02117 and DCR 85-01277, and by the Air Force Office of Scientific Research, under Grant Number AFOSR 89-0186.

[Kohlbecker86]. It is described in Section 3. Section 4 compares the notation used by Ungar and Smith with ours. Section 5 discusses the actual implementation. In the last section we present our conclusions.

2 The Self language

We have chosen to concentrate our implementation on three aspects of the Self language.

In Self there are no identifiers. Instead all information is stored in the method of some object and is accessed by sending that object a message. Because message passing is the primary operation, the language is simplified. An additional result is that the distinction between values that are stored and values that are computed disappears. In one case, however, identifiers are allowed. One of the goals of embedding the language in Scheme is to allow the result to access some of the features provided by Scheme and provided by other systems implemented in Scheme. Therefore in our implementation identifiers refer to Scheme variables. Often objects are globally defined and are accessed using Scheme variables, whereas information local to an object is accessed by sending it messages.

The Self language does not contain classes. Instead every object is an instance. Every object can also be copied, or *cloned* to produce a new instance of that object. In our implementation the new instance is a copy of the original as it was created. It does not reflect assignments made to the variables of the instance. Thus making copies from two different copies of some object will produce identical new objects.

The third feature of Self included in our implementation is the way in which a method is evaluated. It is first cloned. The copy can be thought of as an activation record since it contains the arguments passed to the method. The parent of the new object depends on its type. If it is a dynamic object the parent is the receiver of the method name. If it is a static object the parent is determined when the object is created. In either case the identity of the parent determines in what scope any code in the method is evaluated. Thus using a dynamic object corresponds to using the dynamic environment, whereas using a static object corresponds to using the lexical environment. This process differs from the fluid facility of a language such as Scheme in that the variables themselves have no control over how they are accessed.

These three features of Self, when combined with the facilities provided by Scheme, produce a powerful, yet simple language.

3 Code Expansion

3.1 Syntax

Some versions of Scheme provide the special form **syntax** for defining syntactic extensions. **Syntax** allows expansions to be specified by patterns, permitting complicated syntactic tranformations to be defined in a readable manner. Below we show the definition of the special form **loop** that repeatedly executes a sequence of expressions.

```
(syntax
 (loop ([n v] ...) exp ...)
 (letrec
  ([top (lambda (n ...)
          exp ...
          (top v ...))])
  (top v ...)))
```

As an example of its use,

```
(loop ([a 6])
 (display a))
```

expands into

```
(letrec ([top
          (lambda (a)
           (display a)
           (top 6))])
 (top 6))
```

When it evaluates, it prints **6** repeatedly.

The first argument to **syntax** is a pattern. The name of the syntactic extension appears leftmost in the pattern. All other symbols are variables, which may match any expression.

List structures must match exactly. For example, the pattern `[x y]` will only match a list of two elements. The list may contain any two elements, including arbitrary lists. The symbol `...` following a subpattern means the subpattern matches zero or more matches of the subpattern. In the example `...` follows the pattern variable `exp`. Thus `exp ...` matches any number of expressions. Similarly, `[n v] ...` matches zero or more lists of length two.

The second argument to `syntax` describes the expansion. Pattern variables may be used freely. When `...` is used in the pattern it is considered to be part of the preceding subpattern. Whenever those patterns or their subpatterns are used in the expansion they must be followed by `....` In our example `exp` is followed in the pattern by `....` Thus wherever `exp` appears in the second argument, it or some enclosing expression must be followed by `....` Similarly, since `[n v]` was followed by `...` both `n` and `v` are followed by `...` in the expansion. Symbols that appear in the second argument that are not pattern variables appear in the result exactly.

3.2 Capturing

The expansion of `loop` introduces a new variable, `top`. `Top` is not a pattern variable so it appears in the result exactly. Those arguments to `loop` that appear in the scope of `top` can use its value, even though in the source program, `top` would appear not to be defined, or to be defined differently. When `top` is used in one of these expressions it is said to *capture* the binding introduced by the expansion.

Capturing may or may not be desirable. Making the user aware of the variables bound inside the syntactic extendsion can provide additional power. In the case of `loop` the user can call `top` in any of the arguments and jump to the top of the loop. For example, in

```
(loop ()
 (display "printed")
 (top)
 (display "not printed"))
```

`(display "not printed")` is never reached. If, however, the variables introduced by the expansion are used in a way other than what was intended, the behavior of the syntactic extension will change. For example, if `top` were assigned in one of the `exp ...`, the call to `top` added by the expansion would use the new value of `top` rather than the value intended by the definer of `loop`. Contrary to what the user probably intended, however, the new value of `top` would not be seen outside.

In the implementation of Self the variables `self` and `ego` are captured by the expansions of `make-code` and `make-object`, respectively. While it is an integral part of the language that these variables should be used, they should never be assigned or rebound. The `syntax` facility has *no* way to enforce this restriction. Instead it must be presented as part of the protocol for using the language.

4 Self Notation

As an example of a Self program we present an implementation of a hash table [Ungar87]. Appendix A contains the original program and its translation into the syntax of our system. Our code can be executed using the procedure definitions in Section 5 and Appendix B. We first introduce the syntax of both languages.

Ungar and Smith's syntax is similar that of Smalltalk. Square brackets enclose objects and blocks, whereas curly braces enclose methods. Object and method arguments are either preceeded by a colon and listed between the vertical bars at the beginning of the definition, or indicated using the keyword notation of Smalltalk. Local declarations also appear between the vertical bars. All variables are implicit messages to self, the receiver of the message.

Our notation is different. Because the language is embedded in Scheme, the syntax is more like that of Scheme. Since methods are procedures they need not be defined textually within the enclosing object. A method accesses its context through the variables `ego` and `self` and through its parent. Thus methods need not be defined within the objects in which they are used.

For a more detailed comparison of the two syntaxes we examine **emptySet**. Figure 1 shows its definition using both systems. For comparison purposes we have added the **show** method to the original definition. It appears in our definition because the array is represented as an object rather than explicitly as a vector.

EmptySet describes an object. In the original notation objects are enclosed by square brackets. In ours an object is created using **make-object**. The result is converted to a dynamic object using the procedure **dynamic**. The procedure **static** is available to make static objects. The difference between static and dynamic objects is described in Section 5.

In the original notation methods are defined by giving the name, followed by "=" followed by the value. In ours **make-object** takes five arguments, each of which is a different category of method. The first is the list of parameters. The next is a list of assigning methods and the variables they affect. The third is a list of local variables and their initial values. The fourth is a list of the methods that can be shared by all copies of the object and the fifth is a list of the methods of which each object has its own copy.

In **emptySet**, **size** and **contents** are local variables. There is no method in **emptySet** to side-effect these variables. In **SetTraits** (See Appendix A.), of which **emptySet** is a child, **size:** and **contents:** are defined. **Size:** is a method that assigns the first variable called **size** found on the inheritance chain. In the original notation this is indicated by the fact that the definition of the **size:** method is a left arrow. The variable affected has the same name as the assigning method, but without the final ":". In our system assigning methods can have any name. The name of the assigning method is the first element of each inner list in the second argument to **make-object** and the affected variable is that list's second element. **Contents:** behaves similarly.

The other method defined in **emptySet** is **show**. It is a sharable method. A method is either a code method or an object method. Code methods are defined using **make-code**. Object methods are defined by first creat-

```
emptySet = [ |
    size = 0.
    contents = #(nil) |
    show = {contents}].

AnEmptySet = {SetTraits emptySet clone}.
```

```
(define emptySet
 (dynamic
  (make-object () ()
   ([size (make-code 0)]
    [contents (make-code (array top 10))])
   ([show
     (icdynamic
      (make-object () () () ()
       ([code
         (make-code
          (vsend (vsend ego 'contents)
           'me))])))])
   ())))

(define AnEmptySet
 (emptySet (SetTraits top)))
```

Figure 1: emptySet in the original syntax and in ours

```
(define icstatic
 (lambda (ego obj)
  (invoke 'code (static ego obj))))

(define icdynamic
 (lambda (obj)
  (invoke 'code (dynamic obj))))
```

Figure 2: icstatic and icdynamic

```
(define top
 (ref
  (lambda (method q)
   (q method))))
```

Figure 3: top

ing an object using `make-object` and either `static` or `dynamic`. The result is then passed to `invoke` along with the method name to run when the method is used. In our system this method name is always `code`. Thus we have defined the procedures `icstatic` and `icdynamic` (See Figure 2.) that convert an object into a method. Code methods correspond to the bodies of blocks in the original notation, whereas object methods are those enclosed by curly braces.

Two variables are used in `show`. In the original implementation variables are implicitly sent to self, the receiver of the message. In our system all messages must be sent explicitly to some object. `Self` is the receiver and `ego` is the enclosing object. Any other accessible object can also be used. Usually we pass variable names to `ego` because generally the variable of the enclosing object or some ancestor is what is really desired. In `show`, the message `contents` is sent to `ego`. The result is an array object. We next send `me` to that object to get the internal representation of the array. `Vsend` is used to dereference the location returned for a variable.

Finally, we construct an `emptySet` object, called `AnEmptySet` (See Figure 1.), by passing `emptySet` a parent. As stated above, the parent of an `emptySet` object is a `SetTraits` object. The parent of the `SetTraits` object is just `top` (See Figure 3), a special object that understands no method names.

5 The Implementation

We now turn to the implementation of the two main types used in Self: objects and code. We first establish some terminology, then discuss how these entities are created, and finally describe how each is used.

An *object* is an association between method names and methods. *Dynamic objects* are those that find their methods in the runtime environment. The methods of *static objects* are determined at compile time. In [Ungar87] dynamic objects are referred to as objects and static objects are referred to as closures.

Methods are either *invoked objects*, or *code*. An invoked object is one that has been provided with a method name to run when the arguments to the object are available. An invoked object is *fully invoked* by passing it the receiver of the method name and some arguments. It then produces a value. A *value* is any Scheme value, including objects and code. Code methods are just Scheme code.

A *box* is any zero argument method that when fully invoked returns a reference to a value. A *variable* is a method name bound to a box. A *setter* is a method that changes the value of its associated variable.

There are two ways of looking at an object. The first way, represented by the variable `id`, allows the object to be copied. This variable is only directly accessible to the system-defined method `clone`. The second allows the methods of the object to be accessed. It is represented by the variable `ego`. `Ego` can be used in user-defined methods.

5.1 Creating Objects

Each object has two categories of methods. *Global methods* are those that are created once and are shared by every copy of the object. Global methods do not have access to the variable `ego` and thus do not have direct access to the enclosing object. For this reason global methods are generally dynamic objects, which will inherit the methods of the enclosing object. Global methods are further divided into two categories: the setters and the user-defined global methods. *Local methods* are those that are recreated for each object. There are three kinds of local methods: arguments, local variables and user-defined local methods. Local variables differ from user-defined local methods in that the expression to which one is bound is evaluated when the object is created and the result is stored in the variable's associated box. All local methods have access to the `ego` variable.

`Make-object` (See Figure 4.) takes these subcategories of methods as arguments and expands them into a call to `build`, the procedure that constructs a new object. The format of the arguments to `make-object` is as follows. The first argument is a list of the

parameters of the object. All subsequent arguments are lists of length two. The first element of each inner list of the next argument is the name of a setter and the second is the variable it affects. In the third argument the first element is a local variable and the second is the method that when fully invoked produces the variable's initial value. The fourth contains the name and method of each user-defined global method and the fifth contains the name and method of each user-defined local method. The user-defined methods are either code methods created using `make-code` or invoked objects. An object is converted into an invoked object by passing it along with a method name to the procedure `invoke` (See Figure 12.). `Invoke` returns a procedure that when passed the receiver of the method name and some arguments runs the method specified by the second argument to `invoke`.

`Build` takes three arguments: the global methods, the local methods, and a clone method. `Make-object` packages its arguments into calls to the procedures that make these categories of methods. Each of these procedures sets up some bindings and then returns another procedure. When this procedure is applied to a method name it checks the bindings. If the method name is bound by one of them the associated method is returned. Otherwise it invokes its second argument, a procedure that knows where to look next for the method associated with this method name. This procedure is known as a *failure continuation.*

`Do-globals` (See Figure 5.) is the procedure that makes the global methods. It creates a method for each setter. Each of these takes one argument, the new value. It looks up the associated variable and side-effects the returned location with the new value. The procedure returned by `do-globals` also associates the user-defined global method names with the user-defined global methods.

`Do-locals` (See Figure 6.) creates the local methods. Because each copy of the object has its own set of local methods, they cannot actually be constructed now. Instead `do-locals` just returns a procedure. When this procedure is passed the identity of the object in

```
(syntax
 (make-code exp ...)
 (lambda (self) exp ...))

(syntax
 (make-object
  args
  ([setter affects] ...)
  ([local iexp] ...)
  ([globalname globaldef] ...)
  ([localname localdef ...] ...))
 (build
  (do-globals
   '(setter ...)
   '(affects ...)
   '(globalname ...)
   (list globaldef ...))
  (do-locals '(local ...)
   'args
   '(localname ...)
   (list (lambda (ego) localdef ...) ...))
  (do-clone
   'args
   '(local ...)
   (list (lambda (ego) iexp) ...))))

(define build
 (lambda (globals make-locals make-clone)
  (lambda (clone-parent)
   (let ([ans
          ((deref
            (make-once
             (lambda (id)
              (let ([clone-method
                     (make-clone
                      id clone-parent)])
               (lambda ()
                (make-once
                 (lambda (ego)
                  (search
                   globals
                   (make-locals ego)
                   clone-method
                   (inherit ego)
                  )))))))))])
    (lookup ans 'clone notfound)))))
```

Figure 4: make-code, make-object, build

which the local methods are to be defined, it creates the local methods. At that time fresh boxes are made for the arguments and the local variables. In addition a box is made for the **parent** method. This method is added to every object to contain the object's parent. It is a code method rather than an invoked object. If it were an invoked object it too would have a **parent** method causing an infinite loop. Because the user-defined local methods are each passed the variable **ego**, they have access to the enclosing object.

The **clone** method is created by the procedure **do-clone** (See Figure 8.). Although the **clone** method is shared by every instance, it is created separately because the arguments it uses are different from those used to construct the other global methods. The arguments to **do-clone** are the parameters, the local variables, and their initial values. It requires two other arguments that are not available to the syntactic extension. Thus like **do-locals**, **do-clone** returns a procedure that when passed these arguments returns the **clone** method. The action of the **clone** method is discussed in the section on cloning below (section 3.2).

These three arguments to **build** are those needed by both static and dynamic objects. The argument distinguishing the two types of object is **clone-parent**, the procedure that determines the parent of new objects. Rather than have separate **build** procedures we abstract out the similarities and produce the procedure **build**. Thus **build** takes the three arguments that are common to both types of object and then returns a procedure whose argument is the procedure that creates the difference. Because Scheme supports higher-order procedures, the values of the first three arguments are not lost. The procedure **static** (See Figure 9.) passes the result of **build** a procedure that returns the stored parent. The procedure **dynamic** on the other hand passes the result a procedure that returns whatever parent it is passed. **Build** now has the information needed to build an object of either type.

Internally an object consists of three basic parts. The variable **id**, which allows it to

```
(define do-globals
 (lambda (setters affected names methods)
  (let ([setterfns
         (map
          (lambda (x)
           (invoke 'code (make-assign x)))
          affected)])
   (lambda (msg q)
    (cond [(member msg setters)
           (rlookup msg setters
            setterfns)]
          [(member msg names)
           (rlookup msg names methods)]
          [else (q msg)])))))
```

Figure 5: do-globals

```
(define do-locals
 (lambda (locals args names method-makers)
  (let ([vfn (lambda (x)
              (invoke 'code
               (make-variable (ref x))))])
   (lambda (ego)
    (let ([parent
           (let ([b (ref '())])
            (lambda (s . v) b))]
          [lclfns (map vfn locals)]
          [argsfns (map vfn args)]
          [methods
           (map (lambda (x) (x ego))
            method-makers)])
     (lambda (msg q)
      (cond [(member msg locals)
             (rlookup msg locals lclfns)]
            [(member msg args)
             (rlookup msg args argsfns)]
            [(member msg names)
             (rlookup msg names methods)]
            [(eq? msg 'parent) parent]
            [else (q msg)])))))))
```

Figure 6: do-locals

```
(lambda ()
 (make-once
  (lambda (ego)
   (search
    globals
    (make-locals ego)
    (make-clone id clone-parent)
    (inherit ego)))))
```

Figure 7:

be copied, the variable **ego**, which allows it to refer to itself, and a procedure that when passed a method name returns the method associated with that method name. The body of the **build** procedure sets up these three things and returns a new object.

Id is bound using **make-once** (See Figure 10.). **Make-once** is a procedure that passes to its argument a location. When the application returns, **make-once** stores the result in that location. Thus any procedures created by the argument to **make-once** have access to the value returned by the application. In this case the procedure in Figure 7 is stored in the variable **id**. Each time this procedure is applied it calls **make-once** on **(lambda (ego) ...)**, thus creating a new object.

The call to **search** (See Figure 11.) returns a procedure that takes a method name and a failure continuation as arguments. This procedure then tries to find the method name in each of its arguments. **Locals-maker** is invoked here to make a new set of local methods for the new object. **Inherit** is a procedure that searches for the method name in the ancestors of the new object. It thus also needs to be passed **ego**. Methods in earlier arguments to **search** overshadow those in later ones. For example, because of the order of the arguments to **search** it is possible to redefine the **clone** method.

The arguments and local variables of the object created so far are uninitialized. Thus the last line of **build** calls the **clone** method to create a new initialized object. We next describe the cloning process.

```
(define do-clone
 (lambda (args locals iexps)
  (lambda (id clone-parent)
   (let ([cln
          (lambda (newparent . values)
           (let* ([ego ((deref id))]
                  [fn (lambda (n v)
                       (set-ref!
                        (send ego n) v))])
            (fn 'parent
             (clone-parent newparent))
            (for-each fn args values)
            (for-each fn locals
             (map
              (lambda (x) ((x ego) ego))
              iexps))
             ego))])
    (lambda (msg q)
     (if (eq? msg 'clone)
         cln
         (q msg)))))))
```

Figure 8: do-clone

```
(define static
 (lambda (ego etc)
  (etc (lambda (x) ego))))

(define dynamic
 (lambda (etc)
  (etc (lambda (x) x))))
```

Figure 9: static, dynamic

```
(define make-once
 (lambda (fn)
  (let ([cell (ref 'garbage)])
   (let ([ans (fn cell)])
    (set-ref! cell ans)
    (ref ans)))))
```

Figure 10: make-once

```
(define search
 (lambda l
  (lambda (method q)
   (letrec ([loop
             (lambda (l)
              (if (null? l)
                  (q method)
                  ((car l) method
                   (lambda (method)
                    (loop (cdr l))))))])
    (loop l)))))

(define inherit
 (lambda (ego)
  (lambda (method q)
   (lookup (vsend ego 'parent)
    method q))))
```

Figure 11: search, inherit

```
(define send
 (lambda (ego method . args)
  (apply (lookup ego method notfound)
   ego args)))

(define lookup
 (lambda (ego method q)
  ((deref ego) method q)))

(define invoke
 (lambda (method id)
  (lambda (self . values)
   (apply
    (lookup
     (apply id self values)
     method notfound)
    self ()))))
```

Figure 12: send, lookup, invoke

5.2 Cloning

A copy of an existing object is made using that object's **clone** method. The arguments to the **clone** method are the parent of the new object and the values to which the parameters of the new object should be bound. There are four steps. First, a new copy of the object is created by invoking the value of the **id** variable. Next, **clone-parent** is applied to the provided parent to determine the actual parent of the new object. This value is stored in the **parent** box. Third, each of the values is stored in the arguments. Finally, the initial expressions to which the local variables are bound are evaluated and the results are stored in the local variables. The fully initialized new object is then returned.

5.3 Using Objects

The procedure **send** (See Figure 12.) is used to access the methods of an object. Its arguments are an object, a method name, and any other arguments for the method. First, **send** uses the procedure **lookup** to find the method associated with the method name. **Lookup** passes the object the method name and the failure continuation provided by **send**. Since this system supports only single inheritance, the failure continuation just prints an error message.

The method returned by **lookup** is then run. Its first argument is the receiver of the message, called "self" as in Smalltalk [Goldberg83]. The rest are the arguments to the method. **Send** passes the method the object to which **send** was applied as the receiver, and the rest of **send**'s arguments as the other arguments of the method.

If the method is code, the Scheme code runs immediately. The process for an invoked object is more complex. First the object is cloned. Next the method name specified by the call to **invoke** is passed to the new object. **Send** cannot be used for this step because the receiver for the clone should be the original receiver of the message, not the clone. The process repeats for the new method name in the new object.

If the method is a box, **send** returns a reference to the value rather than the value itself. Returning a reference allows the user to create his own setters.

6 Conclusion

We have shown that it is possible to embed the Self language in Scheme. This embedding is achieved using a small number of syntactic extensions and procedures and was thus much simpler to write than a compiler. Since most of the work is done by the procedures there is only a small increase in code size. Because the Scheme code can be compiled the result should be faster than an interpreter.

We have used Self as an example of a simple and well organized language. We believe, however, that this embedding technique is applicable to a much wider range of programming languages.

Acknowledgements

We are grateful to Stan Jefferson for his discussions with us about object-oriented programming.

References

[Adams 88] Norman Adams and Jonathan Rees, "Object-Oriented Programming in Scheme", in *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, July 1988, 277-288.

[Dybvig87] R. Kent Dybvig, *The Scheme Programming Language*, Prentice-Hall, 1987.

[Felleisen85] Matthias Felleisen, "Transliterating Prolog into Scheme", *Technical Report Number 182*, Indiana University, Computer Science Department, 1985.

[Goldberg83] Adele Goldberg and David Robson, *SmallTalk-80™: The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading MA, 1983.

[Kohlbecker86] Eugene Edmund Kohlbecker Jr, *Syntactic Extensions in the Programming Language Lisp*, PhD thesis, Indiana University, August 1986.

[Meyer88] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall series on Computer Science, 1988.

[Rees86] Jonathan Rees and William Clinger, editors, "The revised[3] report on the algorithmic language Scheme", in *ACM SIGPLAN Notices* 21(12), ACM, December 1986.

[Reynolds72] J.C. Reynolds, "Definitional interpreters for higher-order programming languages", in *Proc. ACM Annual Conference*, 1972, 717-740.

[Ungar87] David Ungar and Randall B. Smith, "Self: the power of simplicity", in *OOPSLA '87 Proceedings*, ACM, 1987, 227-242.

Appendices

A The Hash Example

A.1 Ungar's code

Below is the hash program in Ungar's notation.

```
[ |
    nil = [ ]
    clone = { <primitive> }.
    SetTraits = [ |
        emptySet = [ |
            size = 0
            contents = #(nil) | ].
        size: ←.
        contents: ←.
        clone = {
            super clone
            contents: contents clone }.
        includes: obj = {
            indexFor: obj
                ifPresent: [true]
                ifAbsent: [ |:unused| false] }.
        add: obj = {
            indexFor: obj
                ifPresent: [ ]
                    ifAbsent: [
                        |:i|
                        contents at: i put: obj.
                        size: size + 1.] }.
```

```
        indexFor: obj
        ifPresent: presentBlock
        ifAbsent: absentblock
            = { |
                hashIndex.
                testBlock = [ |:i. c|
                    c: (contents at: i).
                    c isNil ifTrue:
                        [absentBlock value: i].
                    c = obj ifTrue:
                        [presentBlock value] ].
                |
                hashIndex: (obj hash bitAnd:
                    contents lastIndex).
                hashIndex
                    to: contents lastIndex
                    do: testBlock.
                contents firstIndex
                    to: hashIndex - 1
                    do: testBlock.
                grow indexFor: obj }
    | ].
    AnEmptySet = {SetTraits emptySet clone}.
| ].
```

A.2 Our Notation

Below is the same program in our notation. The procedures in Figure 1 should be added to this code.

AnEmptySet is ready to receive messages using **send**. The message **add** adds a new element to the hash table. Its argument is the value to be added. The message **show** returns the current hash table. **Includes** tests whether its argument is in the hash table. Each element of the hash table must be a number.

```
; the methods for an array object

(define at
 (icdynamic
  (make-object (where) () () ()
   ([code
     (make-code
      (vector-ref (vsend ego 'me)
      (vsend ego 'where)))]))))

(define atput
 (icdynamic
  (make-object (where what) () () ()
   ([code
     (make-code
      (vector-set!
       (vsend ego 'me)
       (vsend ego 'where)
       (vsend ego 'what)))]))))

(define array
 (dynamic
  (make-object (lastIndex) ()
   ([me (make-code
         (make-vector
          (vsend ego 'lastIndex) '()))])
   ([at at] [atput atput]) ())))

; the methods for SetTraits

(define includes
 (icdynamic
  (make-object (obj) () () ()
   ([code
     (make-code
      (send ego 'ifipia
       (vsend ego 'obj)
       (icstatic ego
        (make-object () () () ()
         ([code (make-code t)])))
       (icstatic ego
        (make-object (i) () () ()
         ([code (make-code #f)])))))]))))

(define add
 (icdynamic
  (make-object (obj) () () ()
   ([code
     (make-code
      (send ego 'ifipia (vsend ego 'obj)
       (icstatic ego
        (make-object () () () ()
         ([code (make-code t)])))
       (icstatic ego
        (make-object (i) () () ()
         ([code
           (make-code
            (send (vsend ego 'contents)
             'atput (vsend ego 'i)
             (vsend ego 'obj))
            (send ego 'size:
             (+ 1 (vsend ego
                   'size))))])))))]))))
```

```
(define retry
 (lambda (ego)
  (icstatic ego
   (make-object () () () ()
    ([code
      (make-code
       ((send (vsend (vsend ego 'parent)
                     'parent)
         'code)
        self
        (+ 1 (vsend ego 'cur))))])))))

(define found
 (lambda (ego)
  (icstatic ego
   (make-object (x) () () ()
    ([code
      (make-code (vsend ego 'x))])))))

(define ifipiacode
 (lambda (ego)
  (make-code
   (send ego 'hashIndex:
    (mod (hash (vsend ego 'obj))
     (vsend (vsend ego 'contents)
      'lastIndex)))
   ((send ego 'loop
     (icstatic ego
      (make-object (cur) () () ()
       ([code (make-code
               (< (vsend ego 'cur)
                  (vsend
                   (vsend ego 'contents)
                   'lastIndex)))])))
     (icstatic ego
      (make-object () () () ()
       ([code
         (make-code
          ((send ego 'loop
            (icstatic ego
             (make-object (cur) () () ()
              ([code
                (make-code
                 (< (vsend ego 'cur)
                    (vsend ego
                     'hashIndex)))])))
            (icstatic ego
             (make-object () () () ()
              ([code
                (make-code
                 (error () "full"))])))))
          self 0))])))))
   self (vsend ego 'hashIndex)))))

(define hash
 (lambda (x)
  (modulo x 10)))

(define loop
 (lambda (ego)
  (icstatic ego
   (make-object (tst else) () () ()
    ([code
      (make-code
       (icstatic ego
        (make-object (cur) () () ()
         ([code
           (make-code
            (if ((vsend ego 'tst)
                 self
                 (vsend ego 'cur))
                (send ego 'testBlock
                 (vsend ego 'cur)
                 (found ego)
                 (retry ego))
                ((vsend ego 'else)
                 self)))])))))])))))

(define testBlock
 (lambda (ego)
  (icstatic ego
   (make-object (i found retry) ([c: c])
    ([c (make-code '())]) ()
    ([code
      (make-code
       (send ego 'c:
        (send (vsend ego 'contents)
         'at (vsend ego 'i)))
       (cond [(null? (vsend ego 'c))
              ((vsend ego 'found) self
               ((vsend ego 'absentBlock)
                self (vsend ego 'i)))]
             [(equal? (vsend ego 'c)
               (vsend ego 'obj))
              ((vsend ego 'found) self
               ((vsend ego 'presentBlock)
                self))]
             [else ((vsend ego 'retry)
                    self)]))])))))
```

```
(define ifipia
 (icdynamic
  (make-object
   (obj presentBlock absentBlock)
   ([hashIndex: hashIndex])
   ([hashIndex (make-code '())])
   ()
   ([testBlock (testBlock ego)]
    [loop (loop ego)]
    [code (ifipiacode ego)])))))

(define SetTraits
 (dynamic
  (make-object ()
   ([size: size]
    [contents: contents])
   ()
   ([includes includes]
    [add add]
    [ifipia ifipia])
   ())))
```

B Other Procedures

All the code in this paper was written in Chez Scheme [Dybvig87]. Except for the use of `extend-syntax`, `error`, and square brackets, it is compatible with the Scheme standard [Rees86]. The definitions in Figure 13 combined with Figure 2, Figure 3, and the definitions in section 5 form a complete Self system. `Syntax` should be defined first. Next the other syntactic extensions can be defined. Then the other definitions can be loaded in any order. The example in Appendix A must be loaded last. The square brackets are used to make the code easier to read. They can be converted to round parentheses if desired.

```
(extend-syntax (syntax)
 [(syntax (a b ...) c)
  (extend-syntax (a)
   [(a b ...) c])])

(define ref
 (lambda (v)
  (cons v #f)))

(define deref
 (lambda (r)
  (car r)))

(define set-ref!
 (lambda (r v)
  (set-car! r v)))

(define make-variable
 (lambda (box)
  (dynamic
   (make-object () () () ()
    ([code (make-code box)])))))

(define make-assign
 (lambda (name)
  (dynamic
   (make-object (new) () () ()
    ([code
      (make-code
       (set-ref! (send ego name)
        (vsend ego 'new)))])))))

(define rlookup
 (lambda (thing in values)
  (if (eq? thing (car in))
      (car values)
      (rlookup thing
       (cdr in) (cdr values)))))

(define vsend
 (lambda x
  (deref (apply send x))))

(define notfound
 (lambda (method)
  (error 'search
   "method ~s not found"
   method)))
```

Figure 13: syntax, ref, deref, set-ref!, make-variable, make-assign, rlookup, vsend, notfound