To be published in: LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991
© 1991 Kluwer Academic Publishers - Manufactured in The Netherlands

SELF: The Power of Simplicity*

DAVID UNGAR† (*ungar@self.stanford.edu*)

Computer Systems Laboratory, Stanford University, Stanford, California 94305

RANDALL B. SMITH† (*rsmith@parc.xerox.com*)

Xerox Palo Alto Research Center, Palo Alto, California 94304

Abstract. SELF is an object-oriented language for exploratory programming based on a small number of simple and concrete ideas: prototypes, slots, and behavior. Prototypes combine inheritance and instantiation to provide a framework that is simpler and more flexible than most object-oriented languages. Slots unite variables and procedures into a single construct. This permits the inheritance hierarchy to take over the function of lexical scoping in conventional languages. Finally, because SELF does not distinguish state from behavior, it narrows the gaps between ordinary objects, procedures, and closures. SELF's simplicity and expressiveness offer new insights into object-oriented computation.

To thine own self be true.
—William Shakespeare

1 Introduction

Object-oriented programming languages are gaining acceptance, partly because they offer a useful perspective for designing computer programs. However, they do not all offer exactly the same perspective; there are many different ideas about the nature of object-oriented computation. In this paper, we present SELF, a programming language with a new perspective on objects and message passing. Like the Smalltalk-80[1] language [6], SELF is designed to support exploratory programming

*This work is partially supported by Xerox, and partially by National Science Foundation Presidential Young Investigator Grant #CCR-8657631, Sun Microsystems, the Powell Foundation, IBM, Apple Computer, DEC, NCR, Texas Instruments, and Cray Laboratories.

†Authors' present address: Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.

This paper is a substantial revision of [20], originally published in *OOPSLA '87 Conference Proceedings* (*SIGPLAN Notices*, 22, 12 (1987) 227-241).

[13], and therefore includes runtime typing (i.e. no type declarations) and automatic storage reclamation.

But unlike Smalltalk, SELF includes neither classes nor variables. Instead, SELF has adopted a prototype metaphor for object creation [2, 3, 4, 8, 10]. Furthermore, while Smalltalk and most other object-oriented languages support variable access as well as message passing, SELF objects access their state information by sending messages to "self," the receiver of the current message. Naturally this results in many messages sent to "self," and the language is named in honor of these messages.

One of the strengths of object-oriented programming lies in the uniform access to different kinds of stored and computed data, and the ideas in SELF result in even more uniformity, which results in greater expressive power. We believe that these ideas offer a new and useful view of object-oriented computation.

Several principles have guided the design of SELF:

Messages-at-the-bottom. SELF features message passing as the fundamental operation, providing access to stored state solely via messages. There are no variables, merely slots containing objects that return themselves. Since SELF objects access state solely by sending messages, message passing is more fundamental to SELF than to languages with variables.

Occam's razor. Throughout the design, we have aimed for conceptual economy:

- As described above, SELF's design omits classes and variables. Any object can perform the role of an instance or serve as a repository for shared information.
- There is no distinction between accessing a variable and sending a message.
- As in Smalltalk, the language kernel has no control structures. Instead, closures and polymorphism support arbitrary control structures within the language.
- Unlike Smalltalk, SELF objects and procedures are woven from the same yarn by representing procedures as prototypes of activation records. This technique allows activation records to be created in the same way as other objects, by cloning prototypes. In addition to sharing the same model of creation, procedures also store their variables and maintain their environment information the same way as ordinary objects, as described in Section 4.

Concreteness. Our tastes have led us to a metaphor whose elements are as concrete as possible [14, 15]. So, in the matter of classes versus prototypes, we have chosen to try prototypes. This makes a basic difference in the way that new objects are created. In a class-based language, an object would be created by instantiating a plan in its class. In a prototype-based language like SELF, an object would be created by cloning (copying) a prototype. In SELF, any object can be cloned.

[1] Smalltalk-80 is a trademark of ParcPlace Systems. In this paper, the term "Smalltalk" will be used to refer to the Smalltalk-80 programming language.

	class–based systems	**SELF: no classes**
inheritance relationships	instance of subclass of	inherits from
creation metaphor	build according to plan	clone an object
initialization	executing a "plan"	cloning an example
one-of-a-kind	need extra object for class	no extra object needed
infinite regress	class of class of class of ...	none required

The remainder of the paper describes SELF in more detail, and concludes with two examples. We use Smalltalk as our yardstick, as it is the most widely known language in which everything is an object. Familiarity with Smalltalk will therefore be helpful to the reader.

2 Prototypes: Blending Classes and Instances

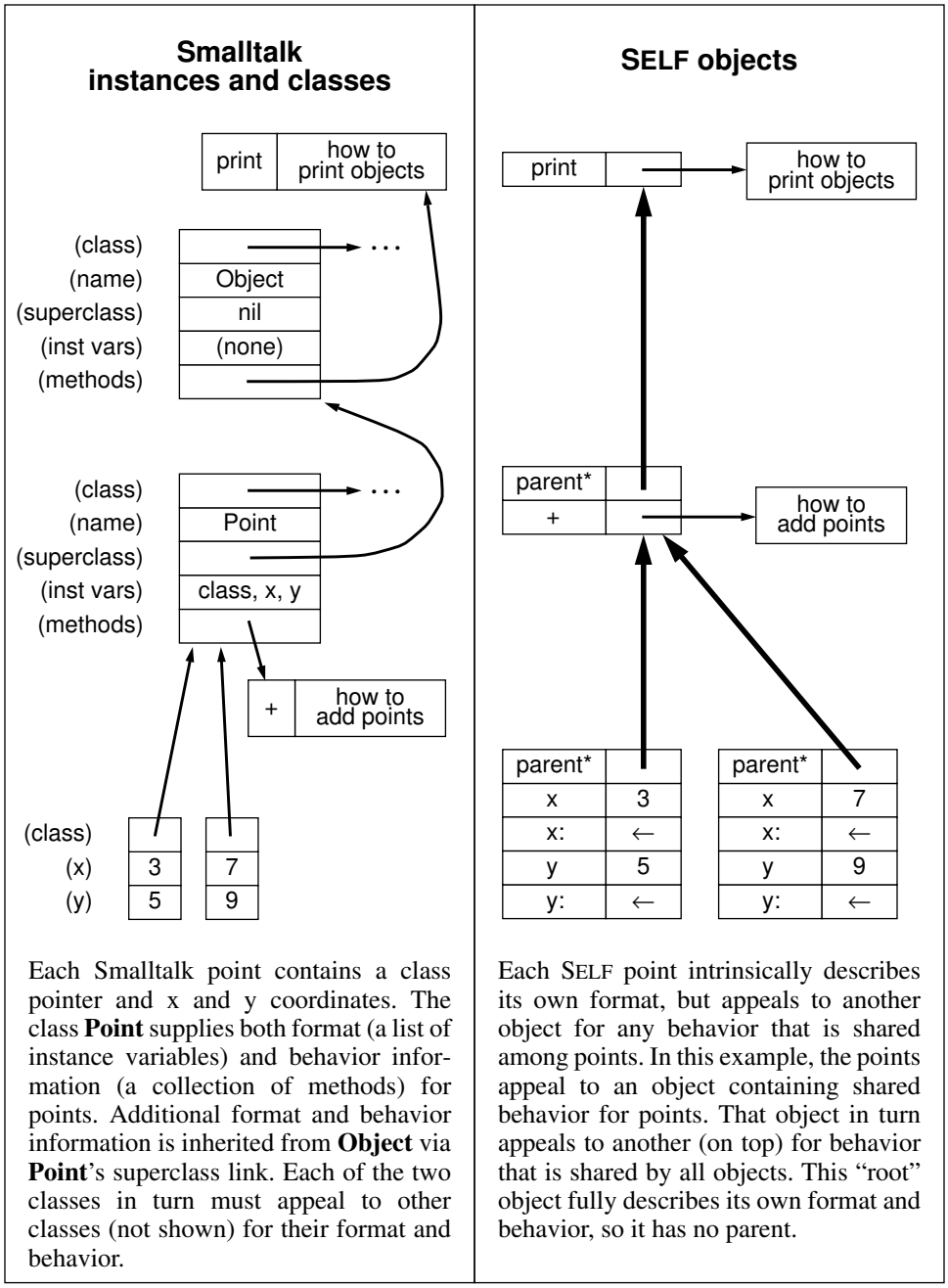
In Smalltalk, unlike C++, Simula, Loops, or Ada, everything is an object and every object contains a pointer to its class, an object that describes its format and holds its behavior (see Figure 1.) In SELF too, everything is an object. But, instead of a class pointer, a SELF object contains named slots which may store either state or behavior. If an object receives a message and it has no matching slot, the search continues via a *parent* pointer. This is how SELF implements inheritance. Inheritance in SELF allows objects to share behavior, which in turn allows the programmer to alter the behavior of many objects with a single change. For instance, a point[2] object would have slots for its non-shared characteristics: x and y (see Figure 1). Its parent would be an object that held the behavior shared among all points: +, –, etc.

2.1 Comparing Prototypes and Classes

One of SELF's most interesting aspects is the way it combines inheritance, prototypes, and object creation, eliminating the need for classes (see above table).

Simpler relationships. Prototypes can simplify the relationships between objects. To visualize the way objects behave in a class-based language, one must grasp two relationships: the "is a" relationship, that indicates that an object is an *instance of* some class, and the "kind of" relationship, that indicates that an object's class is a *subclass of* some other object's class. In a system such as SELF with prototypes instead of classes, there is only one relationship, "inherits from," that

[2]Throughout this paper we appeal to point objects in examples. A Smalltalk point represents a point in two-dimensional Cartesian coordinates. It contains two instance variables: the x and y coordinates.



Each Smalltalk point contains a class pointer and x and y coordinates. The class **Point** supplies both format (a list of instance variables) and behavior information (a collection of methods) for points. Additional format and behavior information is inherited from **Object** via **Point**'s superclass link. Each of the two classes in turn must appeal to other classes (not shown) for their format and behavior.

Each SELF point intrinsically describes its own format, but appeals to another object for any behavior that is shared among points. In this example, the points appeal to an object containing shared behavior for points. That object in turn appeals to another (on top) for behavior that is shared by all objects. This "root" object fully describes its own format and behavior, so it has no parent.

Figure 1. A comparison of Smalltalk instances and classes with SELF objects.
At the bottom of each figure are two point objects that have been created by a user program.

describes how objects share behavior and state. This structural simplification makes it easier to understand the language and easier to formulate an inheritance hierarchy.

A working system will provide the chance to discover whether class-like objects would be so useful that programmers will create them without encouragement from the language. The absence of the class-instance distinction may make it too hard to understand which objects exist solely to provide shared information for other objects. Perhaps SELF programmers will create entirely new organizational structures. In any case, SELF's flexibility poses a challenge to the programming environment; it will have to include navigational and descriptive aids.[3]

Creation by copying. Creating new objects from prototypes is accomplished by a simple operation, copying, with a simple biological metaphor, cloning. Creating new objects from classes is accomplished by instantiation, which includes the interpretation of format information in a class (see Figure 2.) Instantiation is similar to building a house from a plan. Copying appeals to us as a simpler metaphor than instantiation.

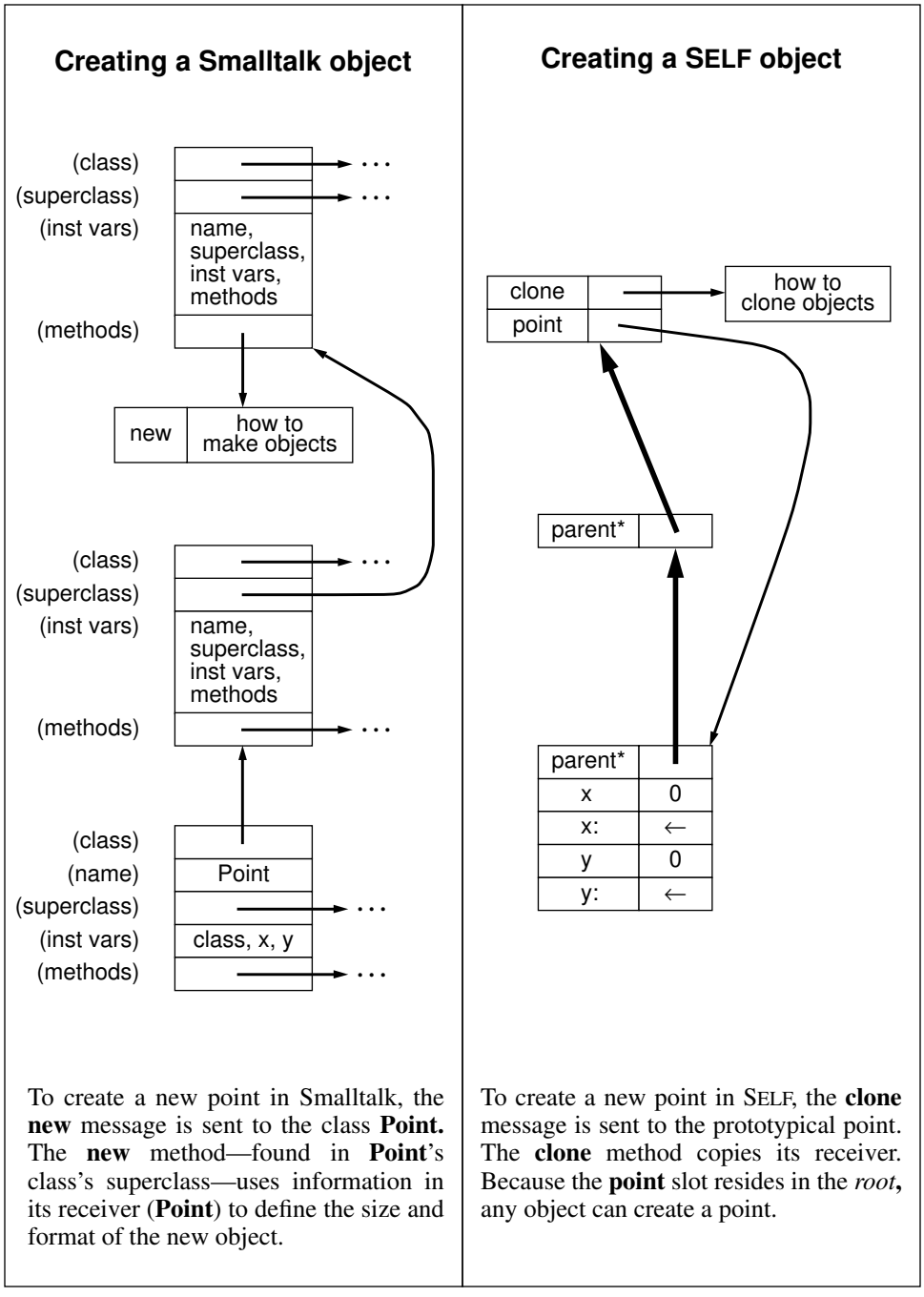
Examples of preexisting modules. Prototypes are more concrete than classes because they are *examples* of objects rather than *descriptions* of format and initialization. These examples may help users to reuse modules by making them easier to understand. A prototype-based system allows the user to examine a typical representative rather than requiring him to make sense out of its description.

Support for one-of-a-kind objects. SELF provides a framework that can easily include one-of-a-kind objects with their own behavior. Since each object has named slots, and slots can hold state or behavior, any object can have unique slots or behavior (see Figure 3.) Class-based systems are designed for situations where there are many objects with the same behavior. There is no linguistic support for an object to possess its own unique behavior, and it is awkward to create a class that is guaranteed to have only one instance. SELF suffers from neither of these disadvantages. Any object can be customized with its own behavior. A unique object can hold the unique behavior, and a separate "instance" is not needed.

Elimination of meta-regress. No object in a class-based system can be self-sufficient; another object (its class) is needed to express its structure and behavior. This leads to a conceptually infinite meta-regress: a point is an instance of class Point, which is an instance of metaclass Point, which is an instance of metametaclass Point, ad infinitum. On the other hand, in prototype-based systems an object can include its own behavior; no other object is needed to breathe life into it. Prototypes eliminate meta-regress.

The discussion of prototypes in this paper naturally applies to them as realized in SELF. Prototype-based systems without inheritance would have a problem: each object would include all of its own behavior—just like the real world—and these

[3] See [19] for a description of the structures that have evolved since this was first written.



To create a new point in Smalltalk, the **new** message is sent to the class **Point.** The **new** method—found in **Point**'s class's superclass—uses information in its receiver (**Point**) to define the size and format of the new object.

To create a new point in SELF, the **clone** message is sent to the prototypical point. The **clone** method copies its receiver. Because the **point** slot resides in the *root***,** any object can create a point.

Figure 2. Object creation in Smalltalk and in SELF.

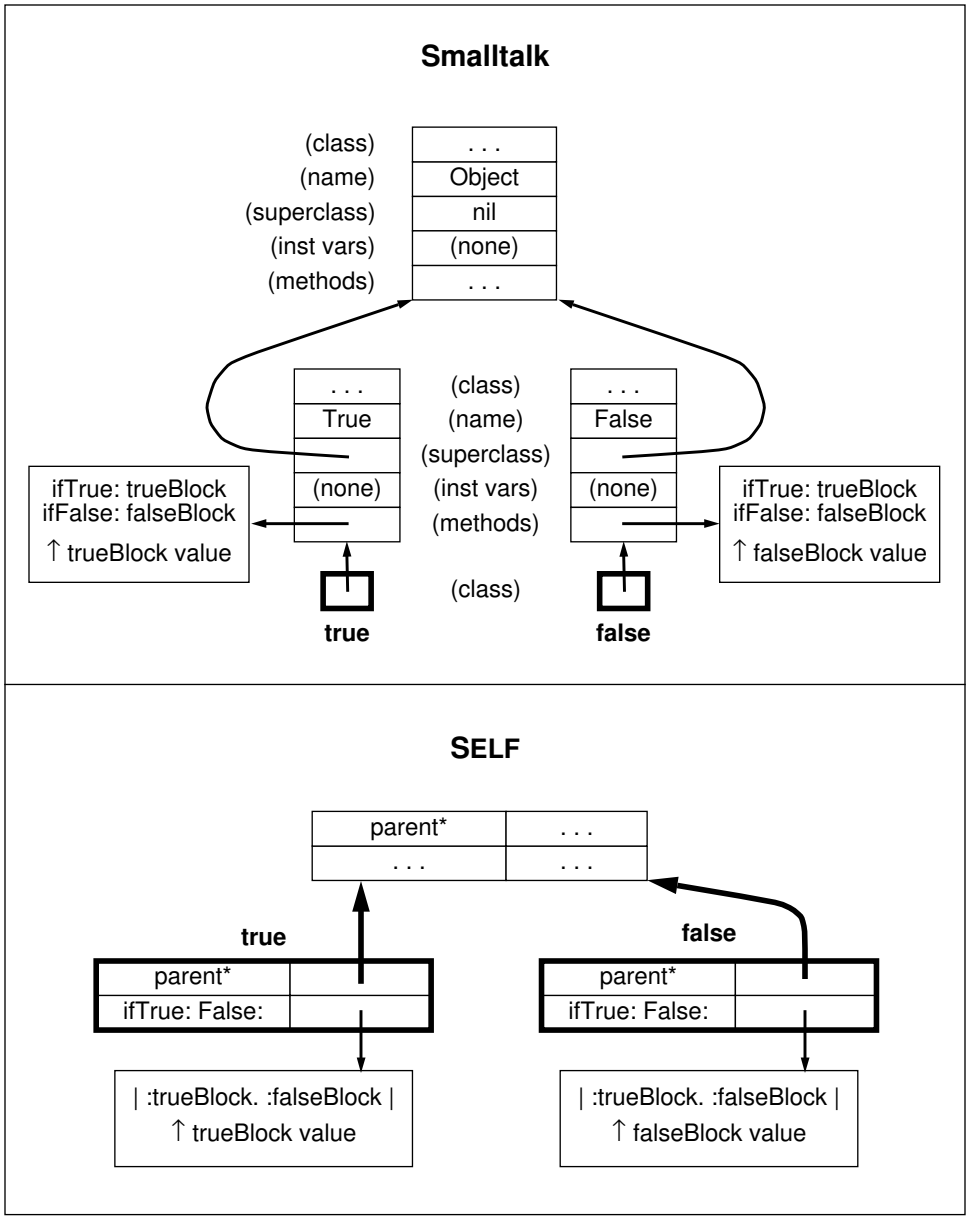


Figure 3. In SELF, it is easier to define unique objects than in a class-based system like Smalltalk. Consider the objects that represent the true and false boolean values. A system needs only one instance of each object, but in Smalltalk, there must be a class for each. In SELF, since any object can contain behavior, it is straightforward to create specialized objects for true and false.

systems would surrender one of the most pleasant differences between computers and the real world, the ability to make sweeping changes by changing shared behavior.

Once inheritance is introduced into the language, the natural tendency is to make the prototype the same object that contains the behavior for that kind of object. For instance, the behavior of all points could be changed by changing the behavior of the prototypical point. Unfortunately, such a system must supply two ways to create objects: one to make an object that is the offspring of a prototype, and another to copy an object that is not a prototype. The ultimate result is that prototypes would become special and not prototypical at all.

Our solution is to put the shared behavior for a family of objects in a separate object that is the parent of all of them, even the prototype. That way the prototype is absolutely identical to every other member of the family. The object containing the shared behavior plays a role akin to a class, except that it contains no information about representation; it merely holds some shared behavior. So to add some behavior to all points in SELF, one would add that behavior to the parent of the points.

3 Blending State and Behavior

In SELF, there is no direct way to access a variable; instead, objects send messages to access data residing in named slots. So, to access its "x" value, a point sends itself the "x" message. The message finds the "x" slot, and evaluates the object found therein. Since the slot contains a number, the result of the evaluation is just the number itself. In order to change contents of the "x" slot to, say, 17, instead of performing an assignment like "x←17," the point must send itself the "x:" message with 17 as the argument. The point object must contain a slot named "x:" containing the assignment primitive. Of course, all these messages sent to "self" would make for verbose programs, so our syntax allows the "self" to be elided. The result is that accessing state via messages in SELF becomes as easy to write as accessing variables directly in Smalltalk; "x" accesses the slot by the same name, and "x: 17" stores seventeen in the slot.

Accessing state via messages makes inheritance more powerful. Suppose we wish to create a new kind of point, whose "x" coordinate is a random number instead of a stored value. We copy the standard point, remove the "x:" slot (so that "x" cannot be changed) and replace the contents of the "x" slot with the code to generate a random number (see Figure 4.) If instead of modifying the "x" slot, we had replaced the "x:" slot with a "halt" method, we would obtain a breakpoint on write. Thus, SELF can express the idioms associated with *active variables* and *dæmons.* Accessing state via messages also makes it easier to share state. To create two points that share the same "x" coordinate, the "x" and "x:" slots can be put in a separate object that is a parent of each of the two points (see Figure 4.)

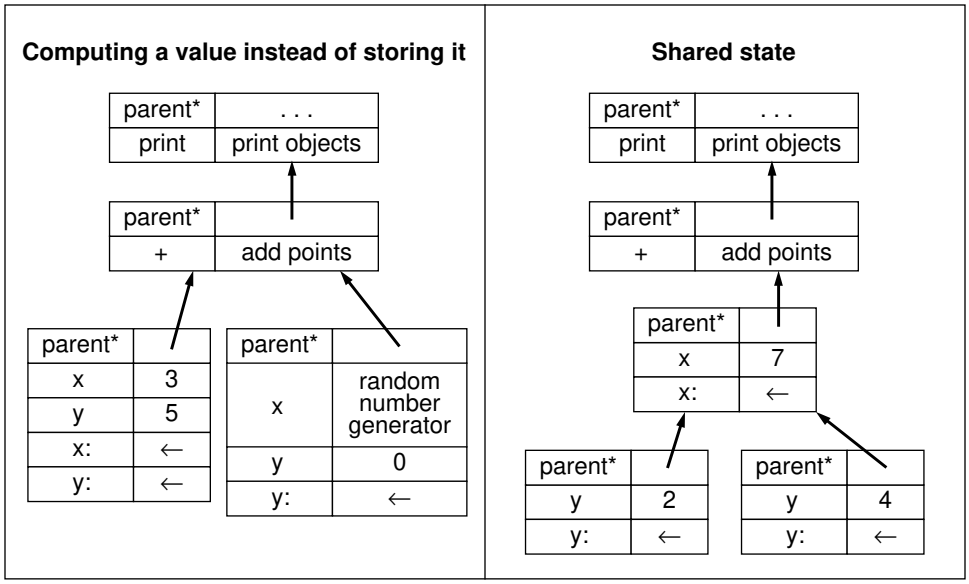


Figure 4. Two examples of flexibility in SELF. On the left is a point whose x coordinate is computed by a random number generator. Since all the code for point sends messages to get the x value, the random x point can reuse all existing point code. On the right are two points that share the same x variable.

In most object-oriented languages, accessing a variable is a different operation than sending a message. This dilutes the message passing model of computation with assignment and access. As a result, message passing becomes less powerful. For instance, the inclusion of variables makes it harder for a specialization (subclass) to replace a variable with a computed result, because there may be code in a superclass that directly accesses the variable. Also, class-based languages typically store the names and order of instance variables in an object's class (as shown in Figure 1). This further limits the power of inheritance; the specification within a class unnecessarily restricts an instance's format. Finally, variable access requires scoping rules, yet a further complication. For instance, Smalltalk has five kinds of variables: local variables (temporaries), instance variables, class variables, pool variables, and global variables, whose scopes roughly correspond to rungs on a ladder of instantiation.

4 Closures and Methods

The Lisp community has obtained excellent results with closures (or lambda-expressions) as a basis for control structures [1, 17]. Experience with Smalltalk blocks supports this; closures provide a powerful, yet easy-to-use metaphor for users to exploit and define their own control structures. Furthermore, this ability is crucial to any language that supports user-defined abstract data types. In SELF, a closure is represented by an object containing an environment link and a method named "value," "value:," "value:With:," and so forth, depending on the number of arguments.

Local variables. Methods require storage for local variables, and in SELF, their slots fulfill this function. In Smalltalk, a method creates an activation record whose initial contents are *described* by the method. For example, the number of temporary variables listed in the method describes the number of fields set aside in the activation record to hold variables. This is similar to the way a class contains a structural description used to instantiate its instances. But in SELF, objects that play the role of methods are *prototypes* of activation records; they are copied and invoked to run the subroutine. So, local variables are allocated by reserving slots for them in the prototype activation record. One advantage is that the prototype's slots may be initialized to any value—they may even contain private methods and closures (blocks).

Environment link. In general, a method must contain a link to its enclosing closure or scope. This link is used to resolve references to variables not found in the method itself. In SELF, instead of having separate scope information, a method's *parent* link performs this function. If a slot is not found in the current scope, lookup proceeds to the next outer scope by following the parent link.

Some interesting mechanisms are needed to make the parent links handle lexical scoping. First, the parent link must be set to the appropriate object. This is simple for an ordinary object; the parent link is automatically set to its prototype's parent, as a result of cloning. For methods, the object created by the compiler serves as a prototype activation, and when invoked, is cloned. The clone's parent then is set to the message's receiver. In this fashion, the method's scope is embedded in the receiver's. For SELF blocks, an environment link is set to the block's enclosing activation record when the block is created. Later, when the block is activated, its method's parent link is set from the block's environment link.

Second, in order to allow the slots containing local variables to be accessed in the same way as everything else, the implicit "self" operand must take on an unusual meaning: start the *message lookup* with the current activation record, but set the *receiver* of the message to be the same as the current receiver. In a way, this is the opposite of the "super" construct in Smalltalk, which starts the lookup with the receiver's superclass (see Figure 5.)

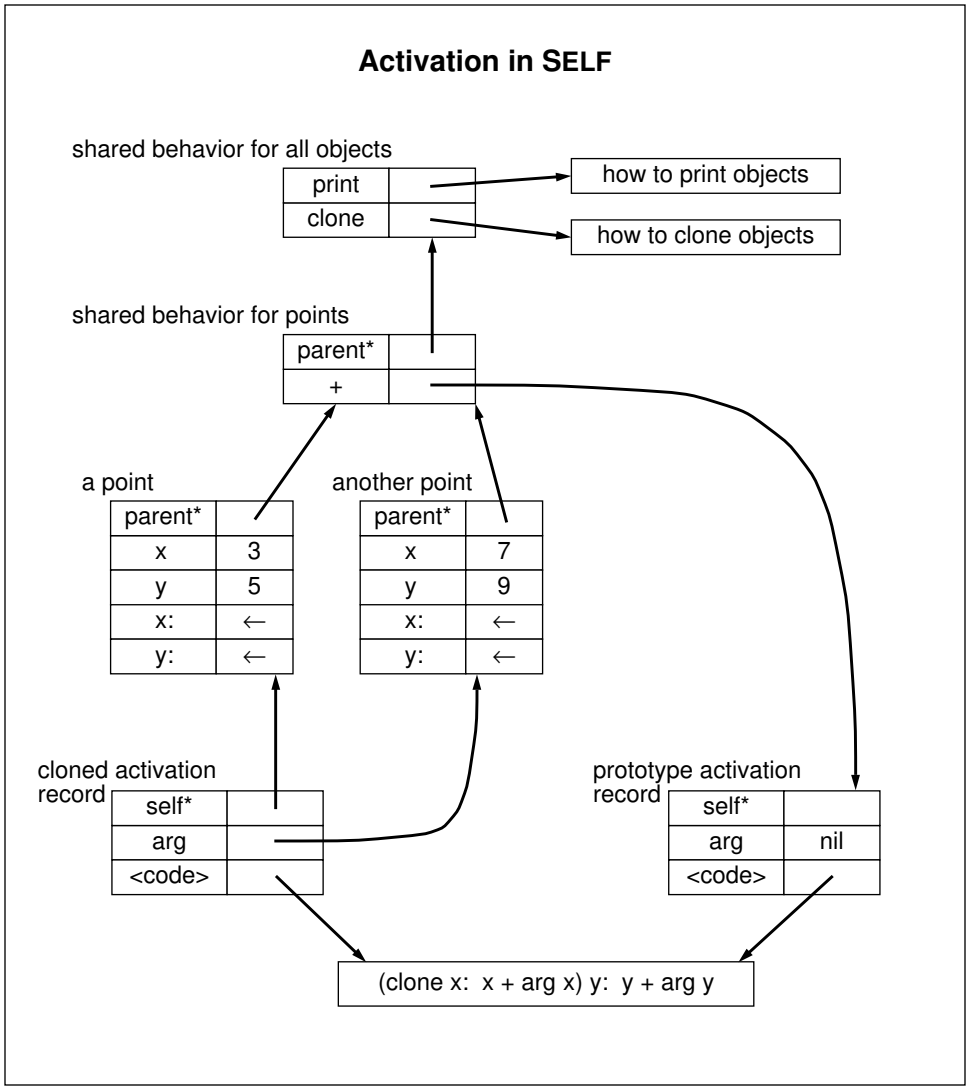


Figure 5. This figure shows what happens when the point (3, 5) is sent the "+" message with argument (7, 9). Lookup for plus starts at (3, 5) and finds a matching slot in the object holding shared behavior for points. Since the contents of the slot is a method object, it is cloned, the clone's argument slot is set to the argument of the message, and its parent is set to the receiver. When the code for "+" executes, the lookup for x will find the receiver's x slot by following the inheritance chain from the current activation record. It will also find the contents of the "arg" slot in the same way. It is this technique of having the lookup for the implicit "self" receiver start at the current activation that allows local variables, instance variables, and method lookup to be unified in SELF.

5 Speculation: Where is SELF Headed?

In designing SELF, we have been led to some rather strange recurring themes. We present them here for the reader to ponder.

Behaviorism. In many object languages, objects are passive; an object is what it is. In SELF, an object is what it does. Since variable access is the same as message passing, ordinary passive objects can be regarded merely as methods that always return themselves. For example, consider the number 17. In Smalltalk, the number 17 represents a particular (immutable) state. In SELF, the number 17 is just an object that returns itself and behaves a certain way with respect to arithmetic. The only way to know an object is by its actions.

Computation viewed as refinement. In Smalltalk, the number 17 is a number with some particular state, and the state information is used by the arithmetic primitives—addition, for example. In SELF, 17 can be viewed as a *refinement of shared behavior* for numbers that responds to addition by returning 17 more than its argument. Since in SELF, an activation record's parent is set to the receiver of the message, method activation can be viewed as the creation of a short-lived *refinement* of the receiver. Likewise, block (closure) activation can be viewed as the creation of a *refinement* of the activation record for the enclosing context scope.

In our examples, we render the shared behavior object for points as an ordinary *passive* object. Another twist would be to build class-like objects out of *methods.* In SELF, the shared behavior object for points could be a method with code that simply returned a clone of the prototypical point. This method could then be installed in the "point" slot of the root object. One object would then be serving two roles: its code would create new points, and its slots (locals) would hold the shared behavior for points. At this writing, we do not believe that this is the best way to construct a system, but the use of methods to hold shared behavior for a group of objects is an example of the flexibility afforded by SELF.

Parents viewed as shared parts. Finally, one can view the parents of an object as shared parts of the object. From this perspective, a SELF point contains a private part with x and y slots, a part shared with other points containing +, -, etc. slots, and a part shared with all other objects containing behavior common to all objects. Viewing parents as shared parts broadens the applicability of inheritance.

6 Syntax

In this section we outline the syntax for a textual representation of SELF objects. Where possible, we have followed Smalltalk syntax to avoid confusion. We have added slot list syntax for creating objects inline. In general, SELF objects are written enclosed in parentheses, and include a list of slots and, in the case of

methods, also include code. Blocks are written as methods enclosed in square brackets instead of parentheses.

The code follows Smalltalk syntax, with a few notable exceptions: The receiver is omitted when it is "self." The return value of a method is always the result of the last expression. Keyword messages associate from right to left. Case is used to make keyword message-sends easier to read: the first keyword must be lowercase, and subsequent keywords in the same selector must be uppercase. These changes reduce the number of parentheses and other lexical noise in SELF code.

The slot list syntax has little precedent in Smalltalk. The slot list, if present, must be nestled in a pair of vertical bars. Each item in the slot list must be separated from the next by a period. (A trailing period is optional.) Finally, there are several forms for slots:

- A selector by itself denotes *two* slots: a slot initialized to *nil,* and a slot named with a trailing colon initialized to the assignment primitive (denoted by ←). For example, the object

  (| **x**. |)

  contains two slots: one called **x** containing nil, and another one called **x:** containing ←. This has the same effect as declaring a Smalltalk variable.
- A selector followed by a left arrow ("<–") and an expression also denotes two slots: a slot initialized to the value of the expression, and a corresponding assignment slot. The effect is similar to an initialized variable. For example, the method object

  ```
  (
     | tally <– 0 |
      10 do: [tally: tally + random].
     tally
  )
  ```

  (where "random" is a slot in the root that returns random numbers) returns the sum of 10 random numbers. It contains a slot named "tally" initialized to zero, and a slot named "tally:" containing the assignment primitive.
- A selector followed by an equals sign ("=") and an expression denotes only one slot, initialized to the value of the expression. The effect is identical to that of the left-arrow form, except that the variable is read-only (no assignment slot is created).
- Finally, a unary selector (i.e. identifier) preceded by a colon defines a slot bound to the corresponding argument of the message. For example: "[| :a. :b | a < b]" defines a block with two arguments, "a" and "b."

The arguments for a method may also be moved into the selector as in Smalltalk:

```
display:At: = (
    | :aForm. :aPoint |
    (bitblt copy
        destination: self At: aPoint Source: aForm)
        copybits)
```

and

```
display: aForm At: aPoint = (
    (bitblt copy
        destination: self At: aPoint Source: aForm)
        copybits)
```

are equivalent.

7 Examples

The first example shows a simple point. It assumes that the system is organized into two kinds of objects: objects that hold shared traits, and prototypes. (In the examples, the primitive _AddSlotsIfAbsent: adds slots to its receiver if the slots do not already exist; the primitive _Define: redefines its receiver in place.)

```
_AddSlotsIfAbsent: ( |
    traits = ().                        Create an object for holding traits objects.
    prototypes* = ().                   Create an object for holding prototypical
| )                                           objects. It is a parent to make it easier
                                              to refer to its contents.
traits _AddSlotsIfAbsent: ( | clonable = () | )
traits clonable _Define: ( |            Define object holding traits of clonable objects.
    copy = ( _Clone ).                  Define a copy method that invokes the
| )                                           "clone" primitive.

traits _AddSlotsIfAbsent: ( | point = () | )
traits point _Define: ( |               Define the object holding traits of points.
    parent* = traits clonable.          Inherit copying methods.
    printString = (                     Define a method to construct a printable string.
        x printString, '@',             Concatenate the strings for x and y,
            y printString ).                separated by an '@'.
    + aPoint = (                        Define method for addition.
        | newPoint |                    Uses a local slot (newPoint).
        newPoint: copy.                 Copy the receiver to serve as the result.
```

```
        newPoint x: x + aPoint x.          Set its x coordinate.
        newPoint y: y + aPoint y.          Set its y coordinate.
        newPoint ).                        Return the new point.
    – aPoint = (                           Define subtraction, exploiting the fact that
        (copy x: x – aPoint x)                 assignment (e.g. x:, y:) returns self.
            y: y – aPoint y ).             (By convention, methods return self if at
| )                                            all possible.)

prototypes _AddSlotsIfAbsent: ( | point = () | )
point _Define: ( |                         Define the prototypical point.
    parent* = traits point.                Inherit shared traits via a read-only parent slot.
    x <– 0.                                Define read/write slots x and y initialized to 0.
    y <– 0.                                (I.e., also define x: and y: with the assignment
| )                                            primitive.)

traits integer _AddSlots: ( |              Add behavior for creating points to existing
    @ y = ( (point copy x: self) y: y ).       integer traits object.
| )
```

The following example is more interesting; it shows how to exploit SELF's unique features to build a data structure that holds a binary tree of objects.

```
traits _AddSlotsIfAbsent: ( | emptyTree = () | )
traits emptyTree _Define: ( |              Define an object holding emptyTree traits.
    parent* = traits clonable.             Inherit copying methods.
    includes: x = ( false ).               Empty trees never include anything
    insert: x = (                          Create a new treeNode, set its contents,
        parent: treeNode copy                  and switch my parents. Uses dynamic
            contents: x ).                     inheritance.
    size = 0.
    do: aBlock = ( self ).
| )

prototypes _AddSlotsIfAbsent: ( | tree = () | )
tree _Define: ( |                          Define the prototypical tree as empty. It has
    parent* <– traits emptyTree.               an assignable parent slot set to
| )                                            traits emptyTree.

traits _AddSlotsIfAbsent: ( | treeNode = () | )
traits treeNode _Define: ( |               Define the object holding tree node traits.
    parent* = traits clonable.             Inherit copying methods.
```

```
    includes: x = ( | subT |                    This method uses a local variable, subT.
        x = contents ifTrue: [^true].           Send contents to self and compare. Uses SELF's
                                                    ability to inherit state (the contents slot).
                                                    Return true if found here.
        subT: x < contents                      Select subtree.
            ifTrue: [left] False: [right].
        subT includes: x ).                     Recurse and return result of recursive call.
    insert: x = ( | subT |                      Insert x into the receiver; same recursion as
        x = contents ifTrue: [^self].               includes:.
        subT: x < contents
            ifTrue: [left] False: [right].
        subT insert: x.
        self ).
    size = ( left size + 1 + right size ).
    do: aBlock = (
        left do: aBlock.
        aBlock value: contents.
        right do: aBlock.
        self ).
    copy = (                                    Copy subtrees when copying a tree node. This
        (resend.copy left: left copy)               provides new empty trees when a tree node
            right: right copy ).                    is copied for insertion into a tree.
| )

prototypes _AddSlotsIfAbsent: ( | treeNode = () | )
treeNode _Define: ( |                           Define the prototypical tree node.
    parent* = traits treeNode.                  A constant parent slot.
    left <– tree.                               Assignable subtree slots, initialized to the
    right <– tree.                                  prototypical (empty) tree.
    contents.                                   Assignable slot for contents.
| )
```

8 Related Work

We would like to express our deep appreciation to the past and present members of the System Concepts Laboratory at Xerox PARC for blazing the trail with Smalltalk [6]. The way SELF accesses state via message passing owes much to conversations with Peter Deutsch, and is reminiscent of an earlier unpublished language of his called "O." Some Smalltalk programmers have already adopted this style of variable accessing [11]. Trellis/Owl is an independently designed object-oriented language that includes syntactic sugar to make message invocation look like

element access and assignment [12]. This is the reverse of our approach. We stuck with message-passing syntax in SELF to emphasize behavioral connotations. Strobe is a frame-based language for AI that also mixes data and behavior in slots [16]. Loops, an extension of InterLisp with objects, also includes active variables [18].

We would like to thank Henry Lieberman for calling our attention to prototypes [10]. Unlike SELF, Lieberman's prototypes include shared information as well. His clones inherit from their prototype, adding private slots on-demand. Although this approach obviates the need for traits objects, its prototypes are heavier-weight objects than SELF's. Exemplars is the name given to prototypes in a project that added a prototype-based object hierarchy to Smalltalk [8]. Like our design for SELF, objects are created by *cloning* exemplars, and multiple representations of the same data type are permitted. Unlike SELF, this system also includes classes as an abstract type hierarchy, and two forms of multiple inheritance. One interesting contribution is the exemplar system's support for or-inheritance. SELF seems to be more unorthodox than exemplars in two respects: it eliminates variable accessing from the language, and it unifies objects, classes, and methods.

The Alternate Reality Kit [14, 15] is a direct manipulation simulation environment based on prototypes and active objects, and it has given us much insight into the world of prototypes. Alan Borning's experience with prototype-based environments, especially ThingLab [2, 3, 4] made him a wonderful sounding board when we were struggling to grasp the implications of prototypes.

The DeltaTalk proposal [5] included several ideas for merging Smalltalk methods and blocks, which helped us to understand the problems in this area. The Actors [7] system has active objects, but these are processes, unlike SELF's procedural model. Actors also rejects classes, replacing inheritance with delegation.

Oaklisp [9] is a version of Scheme with message passing at the bottom. However, Oaklisp is class-based and maintains the inheritance hierarchy separately from the lexical nesting; it does not seem to integrate lambdas and objects.

9 Conclusions

SELF offers a new paradigm for object-oriented languages that combines both simplicity and expressiveness. Its simplicity arises from realizing that classes and variables are not needed. Their elimination banishes the metaclass regress, dispels the illusory distinction between instantiation and subclassing, and allows for the blurring of the differences between objects, procedures, and closures. Reducing the number of basic concepts in a language can make the language easier to explain, understand, and use. However, there is a tension between making the language simpler and making the organization of a system manifest. As the variety of constructs decreases, so does the variety of linguistic clues to a system's structure.

Making SELF simpler made it powerful. SELF can express idioms from traditional object-oriented languages such as classes and instances, but can go beyond them to express one-of-a-kind objects, active values, inline objects and classes, and overriding instance variables. We believe that contemplation of SELF provides insights into the nature of object-oriented computation.

10 Acknowledgments

We would like to thank Daniel Weise and Mark Miller for listening patiently and tutoring us on Scheme. Craig Chambers, Martin Rinard, and Elgin Lee have helped distill and refine the language. Finally, we would like to thank all the readers and reviewers for many helpful comments and criticisms, especially Dave Robson, who helped separate the wheat from the chaff.

References

1. Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs,* MIT Press (1984).
2. Borning, A. H. *ThingLab—A Constraint-Oriented Simulation Laboratory.* Ph.D. dissertation, Stanford University (1979).
3. Borning, A. H. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. In *ACM Transactions on Programming Languages and Systems*, 3, 4 (1981) 353-387.
4. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.
5. Borning, A., and O'Shea, T. DeltaTalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80™ Language. Unpublished manuscript (1986).
6. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).
7. Hewitt, C., and Agha, G. ACTORS: A Conceptual Foundation For Concurrent Object-Oriented Programming. Unpublished draft, MIT Artificial Intelligence Laboratory (1987).
8. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.

9. Lang, K. J., and Pearlmutter, B. A. Oaklisp: An Object-Oriented Scheme with First Class Types. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 30-37.

10. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.

11. Rochat, R. In Search of Good Smalltalk Programming Style. Technical Report No. CR-86-19, Computer Research Laboratory, Tektronix Laboratories, Beaverton, OR (1986).

12. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.

13. Sheil, B. Power Tools for Programmers. *Datamation*, 29, 2 (1983) 131-144.

14. Smith, R. B. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proceedings of 1986 IEEE Computer Society Workshop on Visual Languages* (1986) 99-106.

15. Smith, R. B. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *Proceedings of the CHI+GI '87 Conference* (1987) 61-67.

16. Smith, R. G. Strobe: Support for Structured Object Knowledge Representation. In *Proceedings of the 1983 International Joint Conference on Artificial Intelligence* (1983) 855-858.

17. Steele, G. L., Jr. Lambda, the Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory (1976).

18. Stefik, M., Bobrow, D., and Kahn, K. Integrating Access-Oriented Programming into a Multiprogramming Environment. *IEEE Software Magazine*, 3, 1 (1986) 10-18.

19. Ungar, D., Chambers, C., Chang, B., and Hölzle, U. Organizing Programs Without Classes. To be published in *Lisp and Symbolic Computation*, 4, 3 (1991).

20. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241.