Self: The Power of Simplicity

David Ungar and Randall B. Smith

David Ungar
CIS, Room 209
Stanford University
Stanford, CA 94305
(415) 725-3713
Ungar@Sonoma.Stanford.edu

Randall B. Smith
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304
(415) 494-4947
RSmith.PA@Xerox.com

To thine own self be true.
—William Shakespeare

Abstract

Self is a new 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.

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* language [GoR83], Self is designed to support exploratory programming [She83], 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 [Bor79, Bor81 Bor86, Lie86, LTP86]. 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 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 principals 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 of shared information.
- There is no distinction between accessing a variable and sending a message.

*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

- 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, procedures, and closures are all woven from the same yarn by representing procedures and closures as *prototypes* of activation records. This technique allows activation records to be created the same way as other objects, by cloning prototypes. In addition to sharing the same model of creation, procedures and closures also store their variables and maintain their environment information the same way as ordinary objects, as described below.

Concreteness. Our tastes have led us to a metaphor whose elements are as concrete as possible [Smi87]. 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.

The remainder of the paper describes Self in more detail, and concludes with an example. 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.

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 its *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 as shown in Figure 1, a point* object would have slots for its non-shared characteristics: x and y. Its parent would be an object that held the behavior shared among all points: +, -, etc.

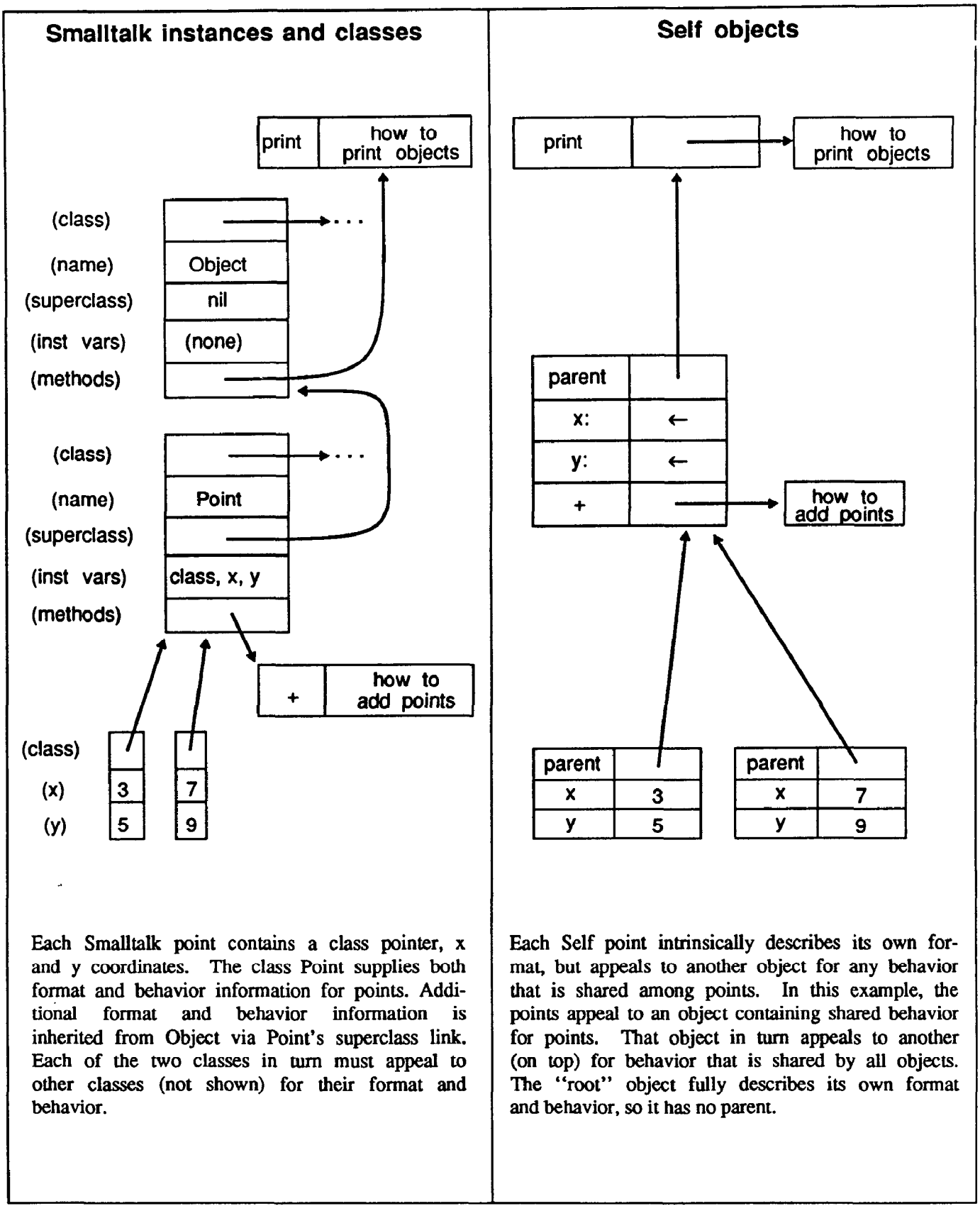
Comparing Prototypes and Classes

One of Self's most interesting aspects is the way it combines inheritance, prototypes, and object creation, eliminating the need for classes.

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 with prototypes instead of classes such as Self, there is only one relationship, "inherits from", that 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

* 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, an x and a y coordinate.



Each Smalltalk point contains a class pointer, x and y coordinates. The class Point supplies both format and behavior information 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. The "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.

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.

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 systems 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.

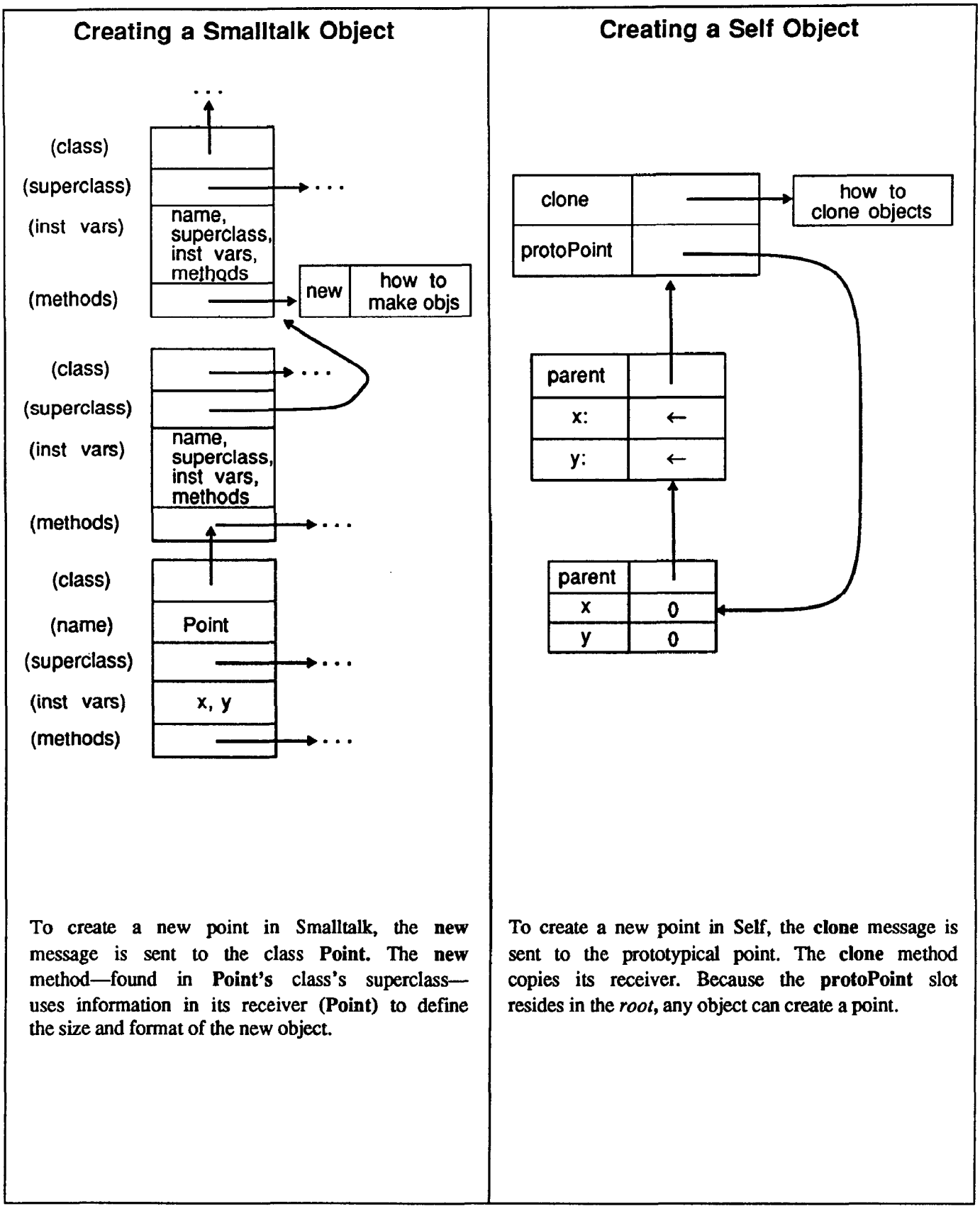
The discussion of prototypes in this paper naturally applies to them as realized in Self. Prototype–based systems without inheritance would have a problem: each object would include all of its own behavior—just like the real world—and these systems would surrender one of the most pleasant differences between computers and the real word, 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. Self avoids these pitfalls by combining prototypes and inheritance.

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 formatting information; 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.

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 (or one of its ancestors) 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



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 **protoPoint** slot resides in the *root*, any object can create a point.

Figure 2. Object creation in Smalltalk and in Self.

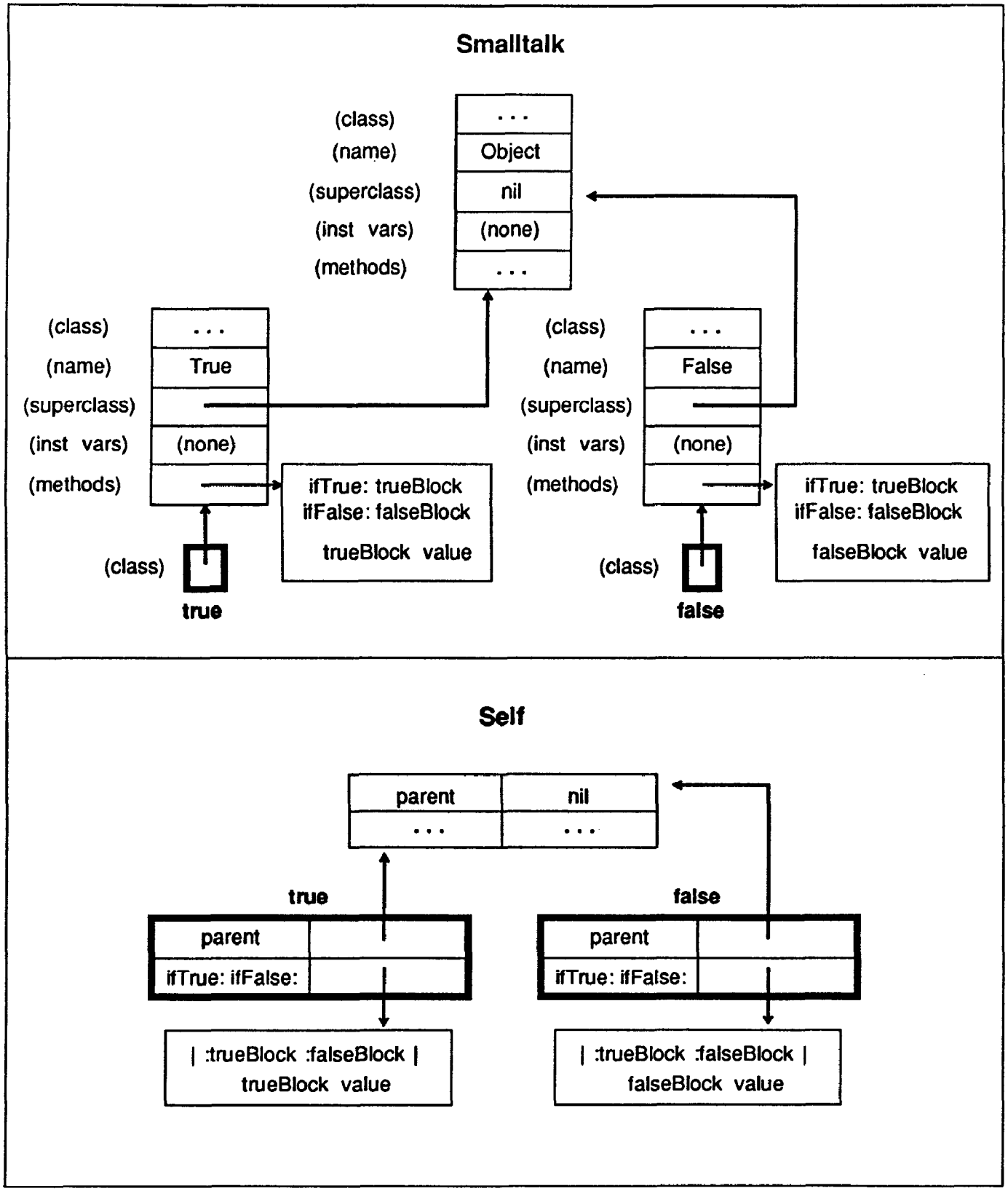


Figure 3. In Self, it is easier to define unique objects than in a class-based system like Smalltalk. Consider the objects that represent the true and false boolean values. A system needs only one instance of each object, but in Smalltalk, there must be a class for each. In Self, since any object can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

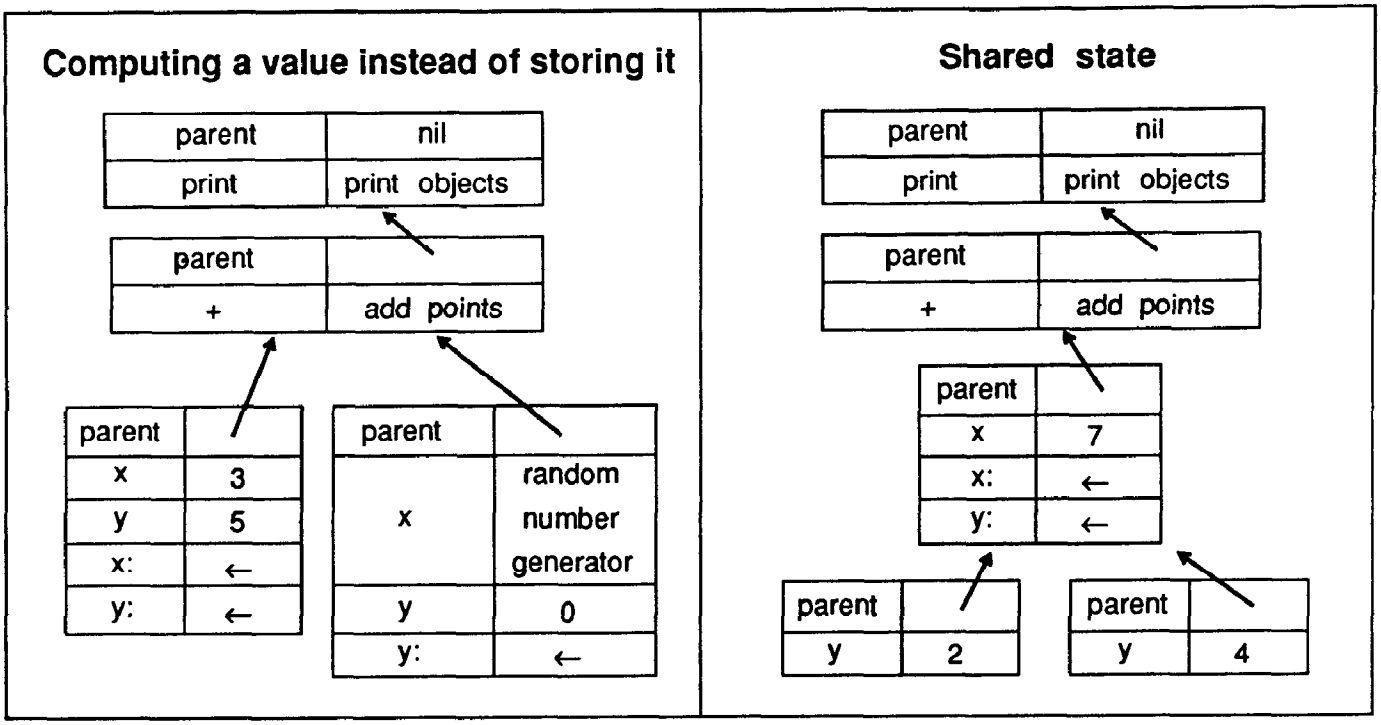


Figure 4. Two examples of flexibility in Self. On the left is a point whose x coordinate is computed by a random number generator. Since all the code for point sends messages to get the x value, the random x point can reuse all existing point code. On the right are two points that share the same x variable.

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 the parents of each of the two points. (Also see Figure 4.)

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 orders 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 the ladder of instantiation.

Closures

The Scheme community has obtained excellent results with closures (or lambda-expressions) as a basis for control structures [Ste76, ASS84]. 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. However, we believe that it is unwise to design a language that makes separate provision for both objects and closures, because they are so similar (both store behavior and state). In Self, objects, closures (blocks) and procedures (methods) have been brought closer together by using slots and inheritance to build closures and procedures:

Local variables. Closures and procedures require storage for local variables, and in Self, their slots fulfill this function. In Smalltalk, invoking a method results in the creation of an activation record whose initial contents is *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 subroutines and closures (methods and blocks) are *prototypes* of activation records; they are copied and invoked to run the subroutine or block. 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—the may even contain private methods and closures (blocks).

Environment link. In general, a closure must contain a link to its enclosing closure or scope. This link is used to resolve references to variables not in the closure itself. In Self, instead of having separate scope information, a closure'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 get set to the appropriate object. This is simple for an ordinary object; the parent link is set to its prototype's parent. For methods (procedures), the object created by the compiler serves as a prototype activation, and when invoked, gets cloned. The clone's parent then gets set to the message's receiver. In this fashion, the method's scope gets embedded in the receiver's. For Self blocks, the parent link must get set to the activation for the enclosing method. This can be done either when the method is activated and the activation record is created, or when the block is created.

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.)

Speculation: Where is Self headed?

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

Behaviorism. In most object languages (Actors excepted), 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 regarding 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 gets 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, or 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.

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 (including methods and blocks) are written enclosed in brackets, and

include a list of slots and some code. Passive objects and blocks are enclosed in square brackets, and methods are enclosed in curly brackets. The code follows Smalltalk syntax, except for the implicit self message destination. The slot list, though, departs from Smalltalk. The first difference, is that the slot list, if present, must be nestled in a pair of vertical bars. Next, 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. If the expression is a Self object with code, the object is treated as a block.

  For example, the method

  ```
  {
          | tally ¨ 0 |
           10 timesRepeat: [tally:
  tally + Random*].
          ^tally
  }
  ```

  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. The effect is similar to an initialized variable.

- 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.

- A keyword (identifier with trailing colon) followed by a left arrow ("¨") defines an *assignment* slot. Such a slot can be used to change the value of a read-only slot elsewhere. For example, points may be defined to be immutable by omitting the assignment slots from them, that is defining the prototypical point as "[| x = nil. y = nil |]." But a routine defined for points can change its receiver's x or y if it includes x: or y: slots.

- Finally, one or more unary selectors (i.e. identifiers) preceded by colons define one slot per identifier, bound to the corresponding argument of the message. For example: "compareBlock = [| :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   destination: self;
                    at: aPoint;
                    source: aForm;
                    copybits }
  ```

  and

  ```
  display: aForm at: aPoint = {
          Bitblt destination: self;
                    at: aPoint;
                    source: aForm;
                    copybits }
  ```

  are equivalent.

*Random is a slot in the root object containing a method that returns a random number.

An Example

The following example shows one way to build a data structure that holds a set of objects. The set is implemented with a open-addressed hash table. A consequence of our notation is that the inheritance hierarchy usually corresponds to the lexical nesting to express (single) inheritance. Here the outermost brackets denote the root object.

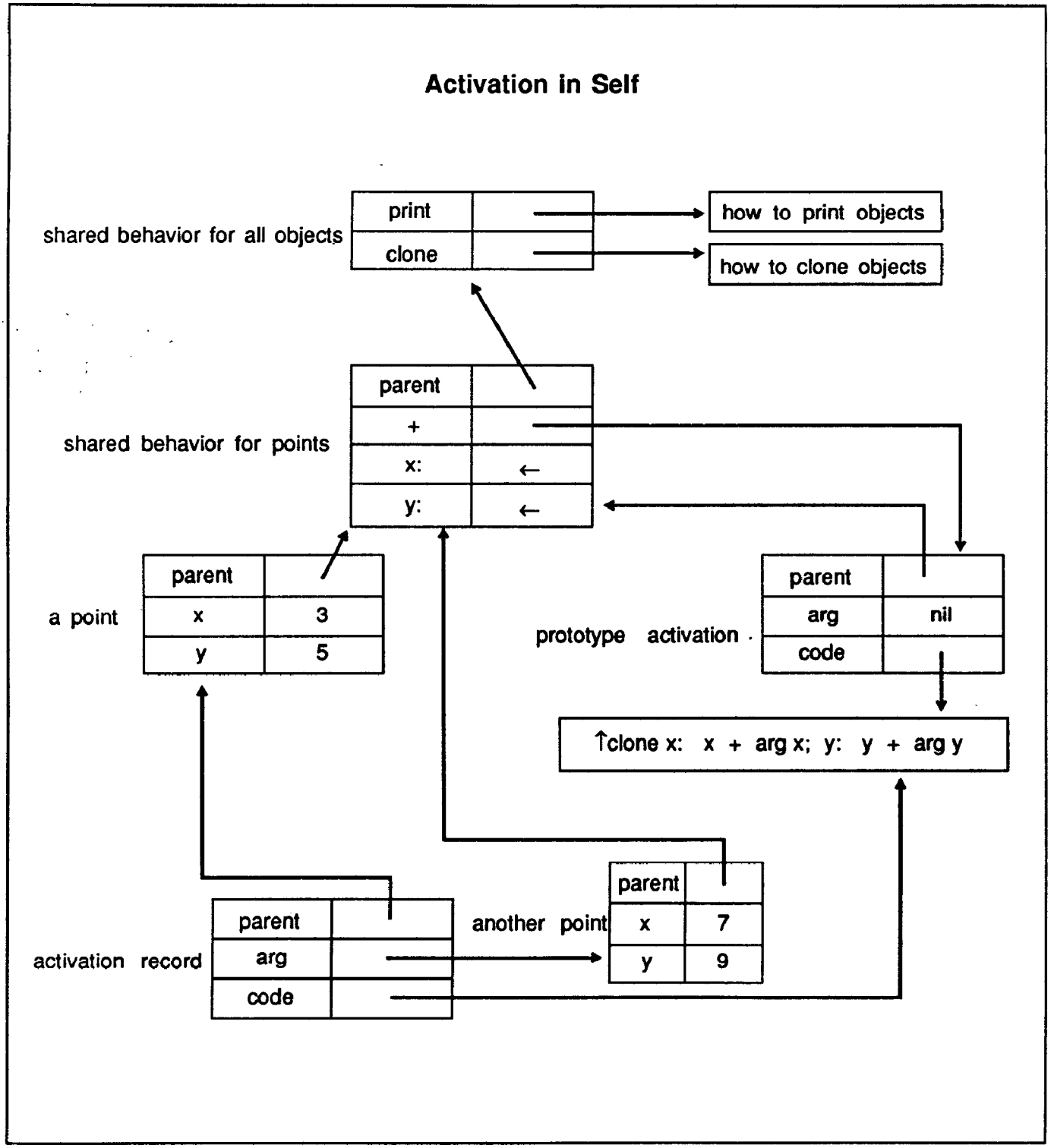


Figure 5. The figure above shows what happens when the point (3, 5) gets sent the plus message with argument (7, 9). Lookup for plus starts at (3, 5) and finds a matching slot in the object holding shared behavior for points. Since the contents of the slot is a method object, it is cloned, the clone's argument slot is set to the argument of the message, and its parent is set to the receiver. When the code for plus 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 args 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.

```
[ |                                                  The global dictionary, or root object.
    nil = [ ].                                       An object with no slots.
    clone = { <primitive> }.                         A method that shallow-copies an object.
    SetTraits = [ |                                  Holds the shared state and behavior for Sets.
        emptySet = [ |                               The prototypical set
            size = 0.                                    with no elements, and
            contents = #(nil) | ].                       contents is an array containing nil.
        size: ".                                     Slots with the assignment primitive,
        contents: ".                                     allowing methods to set the slots of sets.
        clone = {                                    A method to clone sets.
             super clone                             Clone the receiver, and set the clone's
            contents: contents clone }.                  contents to be a clone of the contents array.
        includes: obj = {                            Does the set include obj?
             indexFor: obj                           Send the receiver
                ifPresent: [true]                        indexFor: ifPresent: ifAbsent:.
                ifAbsent: [ | :unused | false] }.    "Unused" is an argument to the block.
        add: obj = {                                 Add obj to the receiver.
            indexFor: obj                            First, test if it's already there.
                ifPresent: [ ]                       It's already there, do nothing.
                ifAbsent: [                          O.K., add it.
                    | :i |                           The block gets passed the index.
                    contents at: i put: obj.         Put it into the array.
                    size: size + 1. ] }.             Increment size.
        indexFor: obj                                This method is privately-used behavior,.
        ifPresent: presentBlock
        ifAbsent: absentBlock                        Search the hash table for obj. If found, return
            = { |                                        the array index where it is. If not found, return
                                                         the index of where it should go. If there is no
                                                         room, enlarge the set.
            hashIndex.                               A read/write slot (local variable)
            testBlock = [ | :i. c |                  TestBlock is a named block (closure) local to this
                                                         method. It takes an array index as its argument,
                                                         called i. It probes the array at that location, and
                                                         if the slot is empty returns the result of
                                                         executing the absentBlock. (AbsentBlock is an
                                                         argument to the enclosing method.) If the slot
                                                         contains the desired object, it returns the result
                                                         from the presentBlock. The explicit returns
                                                         return from the outer method, since this is a
                                                         block. Otherwise, the block just does a local
                                                         return.
                c: (contents at: i).                 Put contents of i^th slot in c.
                c isNil ifTrue:                      If empty execute absentBlock with argument i.
                    [ absentBlock value: i].
                c = obj ifTrue:                      If found, return value of presentBlock.
                    [ presentBlock value]].          End of testBlock.
            |
            hashIndex: (obj hash bitAnd:             Code for indexFor:ifPresent:ifAbsent:.
                contents lastIndex).
            hashIndex                                Use open addressing; search from initial guess to end.
                to: contents lastIndex
                do: testBlock.
```

```
            contents firstIndex
                to: hashIndex - 1
                do: textBlock.
            grow indexFor: obj }


    |].
    AnEmptySet = {SetTraits emptySet clone }.
|].
```

Now search from start to initial guess.

If control falls through, then the receiver is full and the object is not in it, so enlarge the set. (Grow is not shown in this example.)
End of SetTraits.
Returns a new Set.
End of Root Object.

Work in Progress

The design of Self remains unfinished in several vital areas: multiple inheritance, private (encapsulated) slots, and activation details:

Multiple inheritance would add more expressiveness, permitting a better factorization of behavior. We are leaning towards an approach in which each object could have multiple parents, and an error would occur if two slots of the same name were ever found in the course of a lookup. Conflicts could be explicitly resolved by supplying a new method that delegated the message. We also would limit the lookup of messages sent to "self" to only those paths including the sending method. That way the destination of a message sent to "self" would be unaffected by siblings in the inheritance graph.

The separation of format from behavior information in Self's object model would help make multiple inheritance work. For example, many languages with multiple inheritance falter when confronted with inheriting two classes that contain an instance variable with the same name. Extra mechanism is required to specify if the instances should contain only one instance variable that is shared by the two parents, or if the instances should contain two instance variables with the same name. An elegant solution exists in Self. If it is desired to merge the instance variables, the prototype merely contains a slot with the appropriate name. If, on the other hand, it is desired to keep them separate, the prototype has neither slot, but instead can have two parents that each have the slot. The lookup rules guarantee that the slots will be accessed by their appropriate parents. (See Figure 6.)

Encapsulation is lacking in the current design; any object can alter the state of any other object. This could be fixed with some technique for achieving the effect of private slots. Smalltalk protects *variables* but not *methods*, so Self's current lack of encapsulation may not be much worse that Smalltalk's. We are considering ways to incorporate encapsulation into Self.

Activation details are needed to tell a simple, consistent story about methods and blocks.

Status

Craig Chambers, Elgin Lee, and Martin Rinard have built a prototype environment for Self including a browser, inspector, debugger, and interpreter. This system is intended to help us gain a deeper understanding of the language and implementation challenges. We have written and run small Self programs in this environment.

Related Work and Acknowledgements

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 [GoR83]. 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 [Roc86]. Trellis/Owl, an independently designed object-oriented language incorporating static type-checking and encapsulation includes syntactic sugar for element access and assignment [SCB86]. However, the syntax resembles field accessing and assignment. We stuck with message-passing syntax in Self to emphasize behavioral connotations. Strobe was a frame-based language for AI that also mixed data and behavior in slots [Smi83]. Loops, an extension of InterLisp with objects, also included active variables [SBK86].

We would like to thank Lieberman for calling our attention to prototypes in [Lie86]. Exemplars is the name given to prototypes in a project that added a

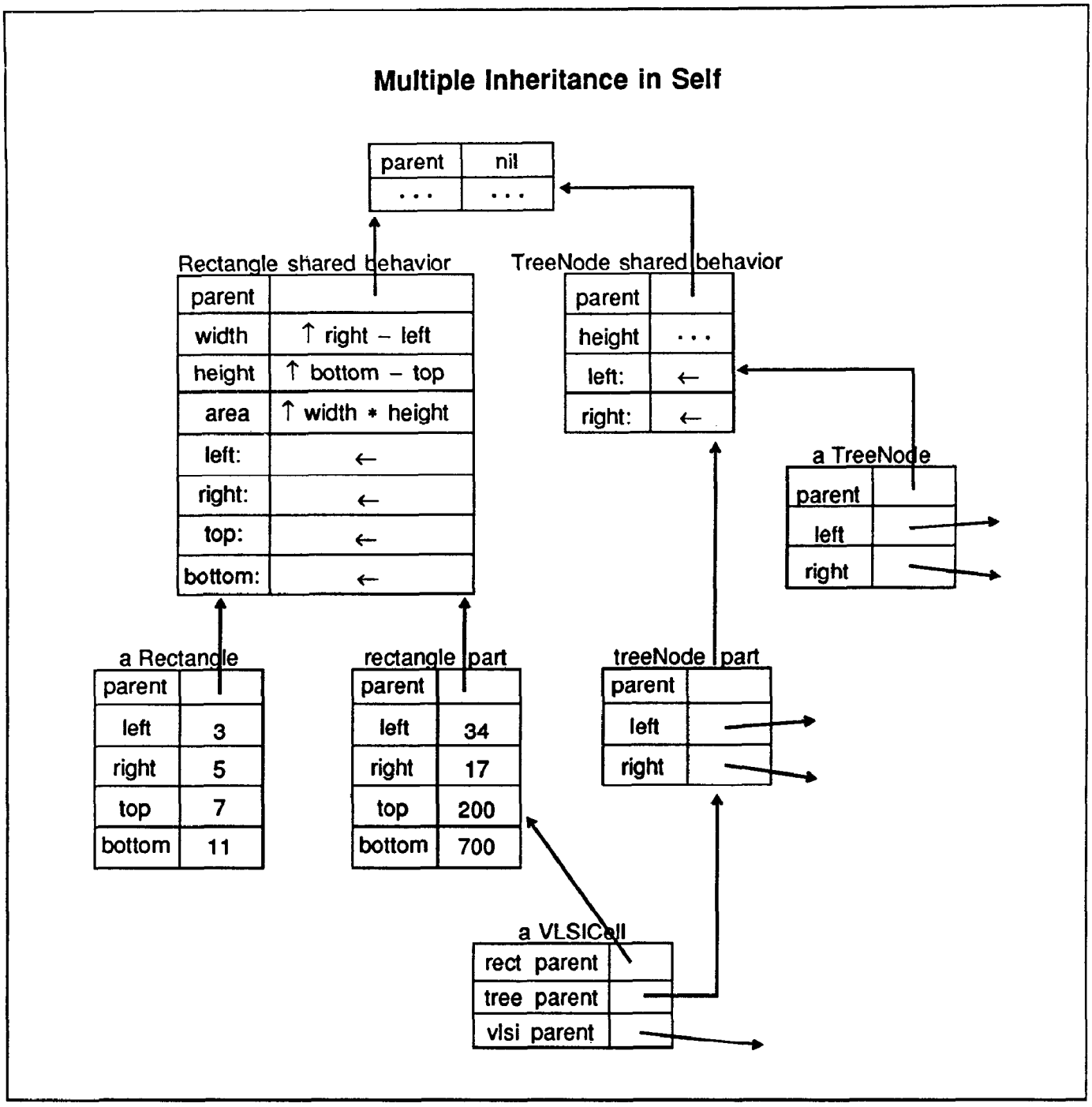


Figure 6. Self's object model is so flexible that it can support multiple inheritance with only small changes. In this example, a VLSI cell object has been created that inherits from both rectangles and trees. The problem for other languages is that, although both rectangles and trees have variables named **left** and **right,** they are used for different purposes and separate slots must be maintained. This can be implemented in Self by creating two extra **parent** objects, identified here as "rectangle part" and "treeNode part" which contain the slots specific to a given inheritance path. When the VLSI cell is sent the **width** message, the lookup will find the rectangle width method, which will in turn send the **right** message to **self.** A special multiple inheritance rule is that messages sent to **self** are looked up only on paths that contain the sender; thus the lookup will find only the **right** slot in the rectangle part—the **right** slot in the "treeNode part" poses no conflict. Self's lack of constraints on an object's formats and parents make this possible.

prototype–based object hierarchy to Smalltalk [LTP86]. Like our design for Self, objects are created by *cloning* exemplars, and multiple representations are permitted. Unlike Self in its present state, 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 and closures.

The Alternate Reality Kit [Smi86] 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 [Bor79, Bor81, Bor86] made him a wonderful sounding board when we were struggling to grasp the implications of prototypes.

The DeltaTalk proposal [BoO86] included several ideas for merging Smalltalk methods and blocks, which helped us to understand the problems in this area. Actors [HeA87] system has active objects, but these are processes, unlike Self's procedural model. Actors also rejects classes, replaces inheritance with delegation.

Oaklisp [LaP86] 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.

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.

This work is partially supported by Xerox, and partially by the National Science Foundation Presidential Young Investigator Award DCK 8657631, NCR, Texas Instruments, and Apple Computer.

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.

References

[ASS84] H. Abelson, G. J. Sussman and J. Sussman, **Structure and Interpretation of Computer Programs,** MIT Press, 1984.

[Bor79] A. Borning, "ThingLab—A Constraint-Oriented Simulation Laboratory," Ph.D. dissertation, Stanford University, March 1979.

[Bor81] A. H. Borning, "The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory," ACM Transactions on Programming Languages and Systems 3,4 (October 1981), 353–387.

[BoO86] A. Borning and T. O'Shea, "DeltaTalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk–80™ Language," unpublished, 1986.

[Bor86] A. Borning, "Classes versus Prototypes in Object–Oriented Languages," Proceedings of the ACM / IEEE Fall Joint Computer Conference, Dallas, TX, November, 1986, 36-40.

[GoR83] A. J. Goldberg and D. Robson, **Smalltalk–80™: The Language and Its Implementation,** Addison-Wesley Publishing Company, Reading, MA, 1983.

[HeA87] C. Hewitt and G. Agha, "ACTORS: A Conceptual Foundation For Concurrent Object-Oriented Programming," MIT AI Lab, January 21, 1987. Unpublished draft.

[LTP86] W. R. LaLonde, D. A. Thomas and J. R. Pugh, "An Exemplar Based Smalltalk," OOPSLA'86 Conference Proceedings, Portland, OR, 1986, 322–330. Also published as a special issue of SIGPLAN Notices Vol. 21, No. 11, Nov. 86.

[LaP86] K. J. Lang and B. A. Pearlmutter, "Oaklisp: An Object-Oriented Scheme with First Class Types," OOPSLA'86 Conference Proceedings, Portland, OR, 1986, 30-37. Also published as a special issue of SIGPLAN Notices Notices, Vol. 21, No. 11, Nov. 86.

[Lie86] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems," OOPSLA'86 Conference Proceedings, Portland, OR, 1986, 214–223. Also published as a special issue of SIGPLAN Notices Notices, Vol. 21, No. 11, Nov. 86.

[RAM84] J. A. Rees, N. I. Adams and J. R. Meehan, **The T Manual (Fourth Edition),** Computer Science Dept., Yale University, New Haven, CT, 1984.

[Roc86] R. Rochat, "In Search of Good Smalltalk Programming Style," Technical Report No. CR-86-19, Computer Research Laboratory, Tektronix Laboratories, Beaverton, OR, 1986.

[SBK86] M. Stefik, D. Bobrow and K. Kahn, "Integrating Access–Oriented Programming into a Multiprogramming Environment," IEEE Software Magazine 3, 1 (January 1986), 10–18.

[SCB86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An Introduction to Trellis/Owl," OOPSLA'86 Conference Proceedings, Portland, OR, 1986, 9-16. Also published as a special issue of SIGPLAN Notices, Vol. 21, No. 11, Nov. 86.

[She83] B. Sheil, "Environments for Exploratory Programming," Datamation, February, 1983.

[Smi83] R. G. Smith, "Strobe: Support for Structured Object Knowledge Representation," Proceedings of the 1983 International Joint Conference On Artificial Intelligence, 1983, 855–858.

[Smi86] R. B. Smith, "The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations," Proceedings of 1986 IEEE Computer Society Workshop on Visual Languages, Dallas, TX, June, 1986, 99–106.

[Smi87] R. B. Smith, "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic," To appear in Proceedings of the CHI+GI'87 Conference, Toronto, Canada, April, 1987.

[Ste76] G. L. Steele Jr., "Lambda, the Ultimate Imperative," AI Memo 353, 1976.

Appendix: Formal Syntax

We herein include a preliminary formal syntax for Self, developed by Craig Chambers from a grammar written by the author for Smalltalk. Terminals are printed in **boldface**, commas separate productions, and double vertical bars indicate alternation.

object = blockObject || methodObject ,

blockObject = [declsPart primitiveSpec body] ,
methodObject = { declsPart primitiveSpec body } ,

declsPart = || | decls optionalPeriod | ,
decls = || decl || decls. decl ,
decl = argsDecl || slotDecl || assignDecl || initDecl ||
constDecl || methodDecl ,

argsDecl = argDecl || argsDecl argDecl ,
argDecl = : IdentifierToken ,
slotDecl = IdentifierToken ,
assignDecl = KeywordToken ¨ ,
initDecl = IdentifierToken ¨ constant ,
constDecl = IdentifierToken = constant ,
methodDecl = slotName = object ,

slotName = selectorsDecl || binarySelector ||
binaryPattern || keywordPattern,
selectorsDecl = selectorDecl || selectorsDecl
selectorDecl ,
selectorDecl = KeywordToken ,
binaryPattern = binarySelector IdentifierToken ,
keywordPattern = KeywordToken IdentifierToken ||
keywordPattern KeywordToken
IdentifierToken ,

primitiveSpec = || < **primitive:** NumberToken >,

body = statements optionalPeriod
optionalReturnStatement ,
optionalPeriod = || . ,
optionalReturnStatement = || expression ,
statements = || expression || statements . expression ,
expression = keywordSend || expression cascadePart ,
cascadePart = ; sendPart ,

sendPart = keywordSendPart || binarySendPart ||
unarySendPart ,
keywordSend = binarySend || implicitSelf
keywordSendPart || binarySend
keywordSendPart ,
keywordSendPart = KeywordToken binarySend ||
keywordSendPart KeywordToken binarySend,
binarySend = unarySend || implicitSelf unarySendPart
|| binarySend binarySendPart ,
binarySendPart = binarySelector unarySend ,
unarySend = primary || implicitSelf unarySendPart ||
unarySend unarySendPart
unarySendPart = IdentifierToken ,

primary = (expression) || explicitSelf ||
explicitSuper || constant ,
implicitSelf = ,
explicitSelf = **self,**
explicitSuper = **super,**
constant = object || scalarConstant || # arrayConstant ,
scalarConstant = NumberToken || StringToken || #
IdentifierToken || CharacterToken || **nil** ||
true || **false** ,
arrayConstant = (arrayElements) ,
arrayElement = IdentifierToken || NumberToken ||
StringToken || CharacterToken ||
arrayConstant ,
binarySelector = BinarySelectorToken || < || > || = ,