The Smalltalk-80 Virtual Machine

Glenn Krasner
Learning Research Group
Xerox Palo Alto Research Center
3333 Coyote Hill Rd
Palo Alto CA 94304

The Smalltalk-80 system is a powerful system that encourages the development of large applications programs. The system contains a compiler, a debugger, a storage management system, text and picture editors, and a file system. It also contains a highly interactive user interface based on graphics that include overlapping windows.

Typically the task of bringing up such a powerful system on a new computer includes writing code to implement these pieces. The Smalltalk-80 system is different in that most of these pieces are written in Smalltalk-80 itself. The part that can be written in Smalltalk-80 is called the *Smalltalk-80 Virtual Image*, and it includes the compiler, debugger, editors, decompiler, and the file system.

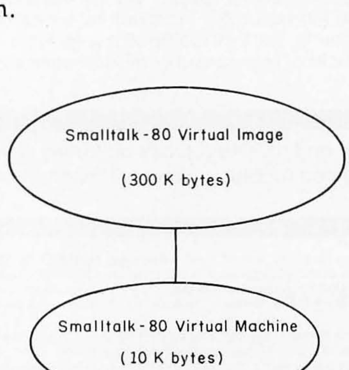

Figure 1: *The Smalltalk-80 Virtual Machine. Most of Smalltalk-80 is written in Smalltalk-80 (the Virtual Image), leaving only a small amount of code that has to be rewritten for each processor on which the language is implemented (the Virtual Machine).*

The remaining part of the Smalltalk-80 system is defined in terms of an abstract machine called the *Smalltalk-80 Virtual Machine* (see figure 1). The Smalltalk-80 compiler translates source code into machine instructions for this virtual machine, rather than translating directly into machine instructions for a particular hardware machine. The task of bringing up a Smalltalk-80 system on a new "target" computer consists only of implementing (writing a program to simulate) the Smalltalk Virtual Machine on the target computer.

In this article, we will present an overview of the elements needed to implement the Smalltalk Virtual Machine. These elements are:

- the *Storage Manager*
- the *Interpreter*
- the *Primitive Subroutines*

Background

A Smalltalk-80 system is made up of *objects* that have state and exhibit behavior. Their state consists of the values of both named and indexed instance variables (which we will call *fields*), and their behavior is exhibited through sending and receiving *messages*. Objects are members of *classes*.

Classes may be *subclasses* of other classes—that is, they may inherit attributes from other classes. Programming in Smalltalk-80 is done by defining the procedures, or *methods*, that are executed when objects receive messages. Typically, messages are sent to other objects to invoke their methods. Sometimes messages invoke *primitive* (machine-code) *subroutines* rather than Smalltalk-80 methods.

From this brief description of Smalltalk-80, we can consider the information needed to implement each of the three elements of the Smalltalk Virtual Machine:

1. To implement the storage manager, we need the information necessary to represent objects in the computer's memory. This information consists of the amount of memory that each object will occupy, which can be computed from the number of fields the object has, and the representation of fields in memory. Objects that describe classes define the number of fields their instances will have, so we also need to know how this number is represented. With this information, we can design a storage manager for objects in a Smalltalk-80 system that will:

- fetch the class of objects
- fetch and store fields of objects
- create new objects
- collect and manage free space

2. The interpreter executes the machine instructions of the Smalltalk-80 Virtual Machine. The information needed to design the interpreter is a description of these machine instructions, called bytecodes (the idea is similar to Pascal p-codes). The bytecodes are contained in methods, so we also need to know the representa-

Who's Who

The design of the Smalltalk-80 Virtual Machine is based on previous Smalltalk systems implemented by the Learning Research Group at Xerox PARC. The original bytecode interpreter design was made for Smalltalk-76 by Dan Ingalls (Ingalls, Dan. "The Smalltalk-76 Programming System: Design and Implementation." In Fifth Annual ACM Symposium on Principles of Programming Languages, *1978, pages 9 through 16). Smalltalk-76 was implemented on the Xerox Alto by Dan Ingalls, Ted Kaehler, Dave Robson, Steve Weyer and Diana Merry, on the Xerox Dolphin by Peter Deutsch, and on the Xerox Dorado by Bruce Horn. TinyTalk was implemented on a Xerox microcomputer by Larry Tesler and Kim McCall (McCall, Kim and Larry Tesler. "Tiny Talk, a Subset of Smalltalk-76 for 64KB Microcomputers." In* Proceedings of the Third Symposium on Small Systems, *ACM Sigsmall Newsletter, Volume 6, Number 2, 1980, pages 197 through 198). Smalltalk-78 (a revised version of Smalltalk-76 similar to Smalltalk-80) was implemented on the Xerox microcomputer by Dan Ingalls, Ted Kaehler, and Bruce Horn, on the Xerox Dorado by Jim Stamos, and on a Norwegian microcomputer (under a research license from Xerox) by Bruce Horn. Smalltalk-80 has been implemented on the Xerox Dorado by Peter Deutsch, on the Xerox Dolphin by Kim McCall, and on the Xerox Alto by Glenn Krasner. The designs of these systems were made by the implementors and other members of the Learning Research Group.*

tion of methods. From this information we can decide how the interpreter will fetch and execute bytecodes and how it will find methods to run when messages are sent.

3. The last piece of information we need to know is which messages will invoke primitive subroutines; that is, which methods we must implement in machine code to terminate the recursion of message sending and to optimize performance.

Before we go into more detail about these elements of a Smalltalk-80 Virtual Machine implementation, here are a few typical figures that will provide a little "reality" to implementors. For the systems that we have implemented at Xerox, the Smalltalk-80 Virtual Image consists of about 300 K bytes of objects. Our typical implementation of the Smalltalk-80 Virtual Machine is 6 to 12 K bytes of assembly code, or 2 K microcode instructions plus 10 K bytes of assembly code. Of this, about 40% is in the storage manager, 20% in the interpreter, and 40% in the primitive subroutines. Our average is about one person-year to implement a fully debugged version of this code.

The Storage Manager

Although the storage manager tends to be the largest and most complex of the three parts of a Smalltalk-80 implementation, the functions it provides are few and relatively simple to understand.

Everything in a Smalltalk-80 system is an object.

Everything in a Smalltalk system is an object, so from a storage point of view memory needs to be divided into blocks, one for each object, plus a pool of memory that is not yet used. Every time a new object is created, a new block of the appropriate size must be found for that object: when objects are no longer used, their memory block may be returned to the pool (see figure 2).

A special entity called an *object pointer* is assigned to each object. If an object pointer were the actual core address of the memory occupied by that object, then there would be fast access to an object given its pointer. However, in the Smalltalk-80 system the object pointer is an *indirect* pointer to the object through a table kept by the storage manager. This allows the storage manager to move an object around in memory without affecting any object that refers to it. It also insures that the storage manager is the only entity in the system concerned with (and allowed to change)

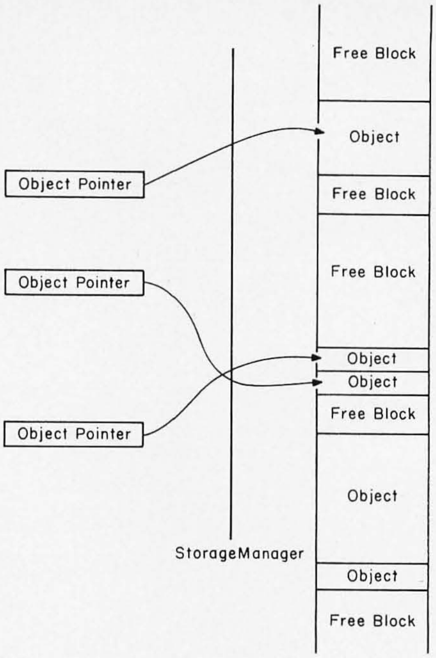

Figure 2: *Objects and memory usage in Smalltalk-80. Each Smalltalk-80 object has an object pointer that points to a block of memory that describes the object. When an object is no longer used, its memory is made available for use.*

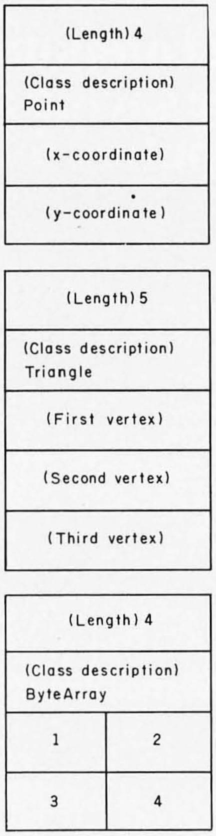

Figure 3: *Typical object representations in Smalltalk-80.*

the actual memory. In the Smalltalk-80 Virtual Image, object pointers are single 16-bit words. This allows for 64 K objects in the system; these objects may take up much more than 64 K words of memory.

Since an object's class and fields are themselves objects, we can see that the block of memory corresponding to an object contains the object pointer of the object's class plus the object pointer for each of the object's fields. The storage manager also keeps the length of the block as one word of the block. This means, for example, that the block corresponding to an object that is an instance of class Point (see figure 3) will have:

- one word that says this block is four words long
- one word that is the object pointer of the object that describes class Point
- one word that is the object pointer of an object that is the *x*-coordinate field of the point
- one word that is the object pointer of an object that is the *y*-coordinate field of the point

Similarly, the block corresponding to an object that is an instance of class Triangle will have:

- one word saying this block is five words long
- one word that is the object pointer of the object that describes class Triangle
- one word that is the object pointer of an instance of class Point, representing one vertex field
- one word that is the object pointer of an instance of class Point, for the second vertex field
- one word that is the object pointer of an instance of class Point, for the third vertex field

For performance optimization, the values in the fields of some objects, such as instances of class ByteArray, will be interpreted as the numerical values themselves, rather than as object pointers. The block corresponding to the byte array containing the elements 1, 2, 3, and 4, in order, will have:

- one word saying this block is four words long
- one word pointing to the object that describes class ByteArray
- one byte encoding the number 1
- one byte encoding the number 2
- one byte encoding the number 3
- one byte encoding the number 4

We will represent all objects as having fields interpreted as object pointers or numerical values, not both. Objects may store numerical values as bytes or words, but not both.

As we have mentioned, the objects that describe classes also need to represent the form of instances of those classes. The essential information is the number of fields the instances will have, and whether these will be pointer or nonpointer fields. For example, the describer of class Point says that its instances will have two fields (x- and y-coordinates) and that these will be pointers (see figure 4). The describer of class ByteArray says that its instances may have a variable number of fields and that these fields will not be pointers but will be numerical values stored in bytes.

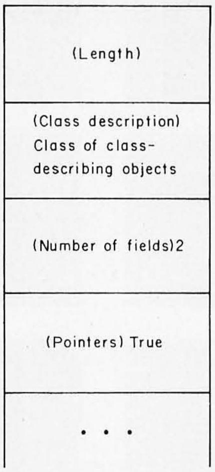

Figure 4: *Class-describing object for class* Point.

The purpose of the storage manager is to fetch and store fields of objects, to create objects, and to manage free space. A clean implementation of the storage manager would be one in which the other parts of the system had access only to the object pointers and made requests of the storage manager only through the following subroutine calls:

- getClass(objectPointer) returns the object pointer of the class of the given object
- getField(objectPointer,fieldOffset) returns the field
- storeField(objectPointer,fieldOffset,newValue) replaces that field with the new value newValue
- newInstance(classObjectPointer,numberOfFields) returns the object pointer of a new instance of that class, and, if that class can have indexed instance variables, this instance has the given number of fields (numberOfFields)

Requests can be made for new storage (with the newInstance subroutine), but not to return used storage. In some other systems, storage that is no longer used must be explicitly returned to the free storage pool. The Smalltalk-80 philosophy is that neither the user nor any part of the system other than the storage manager need have such concerns. Therefore the storage manager must know which objects are no longer being used, so that their storage may reenter the free pool. Typically, Smalltalk-80 Virtual Machine implementations use *reference-counting* to accomplish this. For every object in the system, the storage manager keeps a count of the number of other objects that point to it. This number will change only during execution of the four storage-manager subroutines. When this count reaches zero, the object's memory block may be reused because there are no references to that object anywhere else in the system.

The Interpreter

The interpreter is that portion of the Smalltalk-80 Virtual Machine that performs the actions described in the bytecodes of methods (ie: the machine code of the Virtual Machine). The information needed to implement the interpreter is the description of the bytecodes, the representation of methods, and the technique to find the method to run when sending a message.

The bytecodes define the Smalltalk-80 Virtual Machine as a stack-oriented machine. Each bytecode represents one of the following actions:

- push an object onto the stack
- store the top of the stack as the value for a variable
- pop the top of the stack
- branch to another bytecode
- send a message using the top few elements of the stack
- return the top of the stack as the value for this method

In the Smalltalk-80 Virtual Machine, each of these actions is realized by one or more bytecodes. Note that pushing, storing, popping, and branching are standard instruction types for any stack machine, that sending a message corresponds to calling a procedure using the top few elements of the stack as arguments, and that returning an object from a method corresponds to returning a value from a procedure. The difference between the Smalltalk-80 Virtual Machine and procedure-based stack machines is in the way the procedure is found. In most procedure-based stack machines the address of a procedure is provided in the *execute procedure* instruction; in the Smalltalk-80 system only the "name," called the *selector*, of the message is provided; the method (or procedure) to be executed is found through a strategy involving the receiver of the message and its class. We will first describe the bytecodes, then how methods are represented, and finally give a strategy for finding methods.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Push 4	(3 4)
-3- Push 5	(3 4 5)
-4- Send +	(3 9)
-5- Send *	(27)

Table 1: *Bytecodes for the Smalltalk expression* 3 * (4 + 5).

Stack Operations

The Smalltalk-80 Virtual Machine and corresponding bytecode set are stack oriented. Object pointers are pushed and popped from a stack, and when a message is sent, the top few elements of the stack are used as receiver and arguments of the method. These are replaced by the object returned as the value of that method. For example, the Smalltalk-80 expression:

3 * (4 + 5)

is encoded by the bytecodes shown in table 1.

As bytecodes labeled -1-, -2-, and -3- are executed by the interpreter, the objects 3, 4, and 5 are pushed onto the stack. When bytecode -4- is executed, the message + is sent to the second object on the stack (4) with the top object of the stack as the argument (5). The 4 and 5 are popped off this stack when the message is sent, and the interpreter begins executing the bytecodes for the method corresponding to the message + in the Smalltalk class of small integers. This method will eventually return an object, in this case 9, as its value, and the interpreter will push the 9 onto the original stack above the 3 and resume execution with bytecode -5-. Bytecode -5- will produce an effect similar to that produced by -4-, leaving the object 27 on the stack. In the same way that other stack machines push *data* onto a *stack* and use the top few data items as *arguments* for a *procedure*, replacing them with the *value* returned from that procedure, the Smalltalk-80 Virtual Machine pushes *object pointers* onto a *stack*

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Push 4	(3 4)
-3- Send +	(7)
-4- Store into a	(7)

Table 2: *Bytecodes for the Smalltalk expression* a ← 3 + 4.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Store into a	(3)
-3- Pop	( )
-4- Push 4	(4)
-5- Store into b	(4)

Table 3: *Bytecodes for the Smalltalk expression* a ← 3. b ← 4.

Bytecode	Stack Contents After Execution (Top of Stack to Right)
-1- Push 3	(3)
-2- Store into a	(3)
-3- Pop	( )
-4- Push a	(3)
-5- Return top of stack	( )

Table 4: *Bytecodes for the Smalltalk expression* a ← 3. ↑ a.

and uses the top few as *receiver* and *arguments* of a *message*, replacing them with the *object* returned from that method.

In both machines, values from the top of the stack may be stored as the values of variables. As an example, the Smalltalk expression:

a ← 3 + 4

will be represented by the bytecodes in table 2. Here, -1-, -2- and -3- act as before and the interpreter executes bytecode -4- by storing the top of the stack 7 into the variable a.

Stack machines in general, and the Smalltalk-80 Virtual Machine in particular, also have the ability to pop the top element off the stack. In the statements:

a ← 3.
b ← 4

once the 3 is stored into variable a, it is no longer needed, so it is popped from the stack. These statements are represented by the bytecodes shown in table 3.

The top of the stack may be returned as the value for the method. The statements:

a ← 3.
↑ a

are represented by the bytecodes shown in table 4.

Branching Operations

Conditional and looping messages are used so often that they are represented not by actual messages but by bytecodes for conditional and unconditional jumps. (This is *only* for performance reasons; these branching and looping messages would work if they were actually sent like other messages.) For example:

a > 4 ifTrue: [a ← a − 1]

(which in the Smalltalk-80 system means execute the code within the brackets only if the object returned from the > message is not false) is represented in table 5 (ignoring the stack from now on).

Bytecode

-1- Push 4
-2- Push a
-3- Send >
-4- Jump to -10- if the top of the stack is false
-5- Push a
-6- Push 1
-7- Send −
-8- Store into a
-9- Pop
-10- < the next bytecode >

Table 5: *Bytecodes for the Smalltalk expression* a > 4 ifTrue: [a ← a − 1].

Bytecode

-1- Push a
-2- Push 4
-3- Send >
-4- Jump to -11- if top of stack is false
-5- Push a
-6- Push 1
-7- Send −
-8- Store into a
-9- Pop
-10- jump to -1-
-11- < the next bytecode >.

Table 6: *Bytecodes for the Smalltalk expression* [a > 4] whileTrue: [a ← a − 1].

Table 6 shows the bytecodes for the looping expression:

[a > 4] whileTrue: [a ← a − 1]

(which means execute the code in the second brackets as long as the code in the first set of brackets evaluates to something other than false).

Addressing Variables

Methods are implemented as objects whose fields contain the bytecodes plus a group of pointers to other objects called the *literal frame*. The interpreter can use the getField subroutine of the storage manager to fetch the next required bytecode to execute. This takes care of returns, jumps, and pops, but for the other bytecodes we need to represent more information. In particular, for the push and store bytecodes, we need to represent where to find the object pointers to push; for the send bytecodes, we need to represent where to find the selector of the message and which stack elements are the receiver and arguments.

The source code for a method contains variable names and literals, but the bytecodes of the Virtual Machine are defined only in terms of field offsets. From the Virtual Machine's point of view, there are three types of variables: variables local to the method (called *temporaries*), variables local to the receiver of the message (*instance variables*), or variables found in some dictionary that the receiver's class shares (*global variables*). Note that *class variables* are treated in the same way as other global variables. The Smalltalk-80 compiler (itself written in Smalltalk-80) translates references to these variables into bytecodes that are references to field offsets of the receiver, the temporary area, or globals. The instance variables are translated using a field of class-describing objects that associates instance variable names with field offsets. The assignment of offsets to temporaries is done when the compiler translates a method by associating names of temporaries to offsets in the temporary area. The compiler creates instances for the literals, puts their object pointers into the literal frame of the method, and produces bytecodes in terms of offsets into the literal frame. For global variables, the compiler uses system dictionaries that associate global names to indirect references to objects. Object pointers of the indirect references to the global objects are also placed in the literal frame of the method. The bytecodes for accessing globals are encoded as indirect references through field offsets of the literal frame.

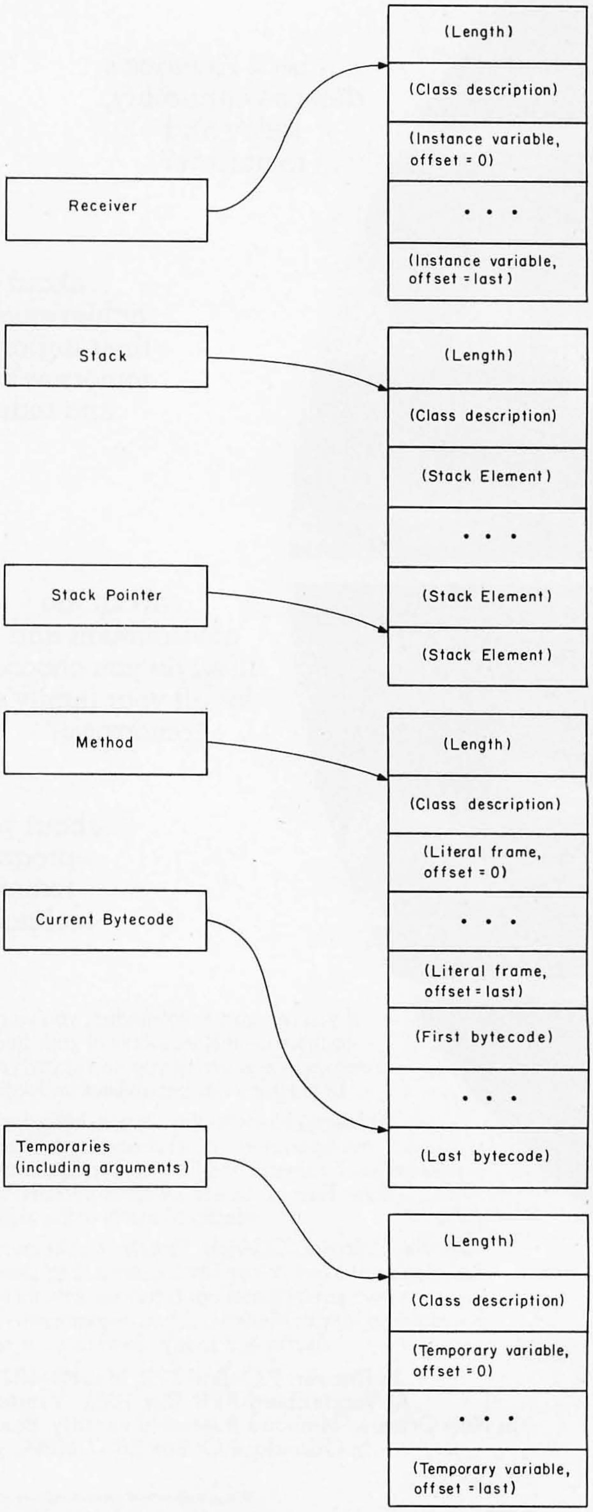

Figure 5: *Object pointers held by the interpreter.*

This means that when the interpreter is executing a method, it has to keep a stack, a temporary area, a pointer to the receiver and arguments of the method, and a pointer to the method itself (see figure 5). It uses the storage manager's getField and storeField subroutines to push and pop pointers from the stack object, to retrieve and set values of variables in the temporary area, to retrieve and set values of variables of the receiver, and to get bytecodes and values of global variables from the method.

Finding Methods

When a message is sent, the receiver and arguments must be identified, and the appropriate method must be found by the interpreter. The technique used in Smalltalk-80 is to include in each class-describing object a dictionary, called the *method dictionary*, that associates selectors with methods. Pointers to the selectors that will be sent by any method are kept in the method (along with global variable pointers and bytecodes). The bytecodes that tell the interpreter to send a message encode a field offset in the literal frame where the selector is found, plus the number of arguments that that method needs. By convention, the top elements of the stack are the arguments and the next one down is the receiver. For example, the send bytecode for the expression:

```
3 + 4
```

will stand for "send the selector in field X of the method (which will be +), and it takes one argument." The interpreter will ask the storage manager for the X field of the method, will get the top of the stack (4) as the argument, and the next element down (3) as the receiver. It will locate the receiver's class, its method dictionary, search it for an association of the + selector with some method, and, when found, execute that method.

If no such association is found, the searching does not end. The receiver's class may be a subclass of another class, called its *superclass*. If this is the case, the method for + may be

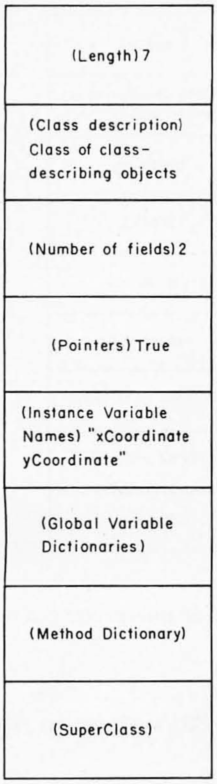

Figure 6: *Class-describing object for class* Point, *revisited.*

defined in the superclass, so the interpreter must check there. This means that each class must have a field that refers to its superclass (see figure 6). The interpreter searches the method dictionary of the superclass, its superclass, and so on, until either an appropriate method is found or it runs out of superclasses, in which case an error occurs.

To execute a method, the interpreter needs a place for temporaries and a stack for that method. In the Smalltalk-80 Virtual Machine, this is done by allocating an object that is an instance of class MethodContext. Objects in MethodContext keep track of the method, the stack for that method, a pointer to the next bytecode to be executed in that method, the temporary variables for that method, and the context from which that method was invoked, called the *caller* of that method (see figure 7). When a method returns, the value returned is pushed on the stack of the caller context, and execution continues at the next bytecode of the caller's method.

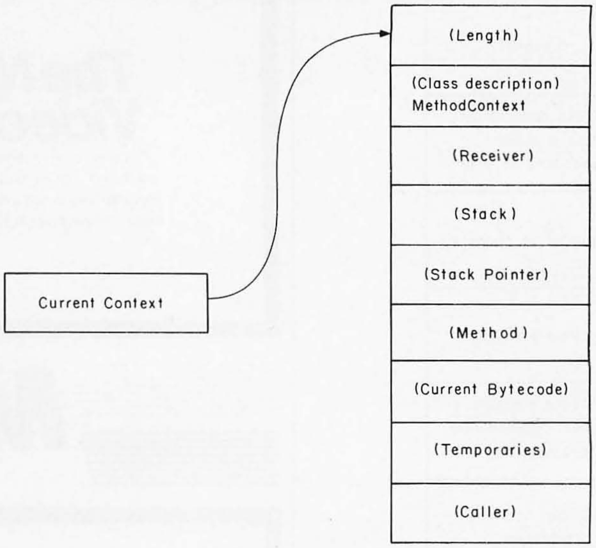

Figure 7: *The only object pointer used by the Smalltalk-80 interpreter is a reference to a* MethodContext.

The Smalltalk-80 Virtual Machine implementation is a program running in the machine language of the target computer.

Primitive Subroutines

The Smalltalk-80 Virtual Machine implementation is a program running in the machine language of the target computer. The storage manager is the collection of subroutines in this program that deals with memory allocation and deallocation. The interpreter is the collection of subroutines in this program, one of which fetches the next bytecode from the currently running method and calls one of the others to perform the appropriate action for that bytecode. In addition to these functions, we have found that there are several other places in the Smalltalk-80 system where performance considerations make it necessary, or at least desirable, to implement certain functions as machine-code subroutines in the Smalltalk-80 Virtual Machine. These places are:

- input/output: connecting the Smalltalk-80 system to the actual hardware
- arithmetic: basic arithmetic for integers
- subscripting indexable objects: fetching and storing indexable instance variables
- screen graphics: drawing and moving areas of the screen bitmap quickly
- object allocation: connecting the Smalltalk-80 code for creating a new instance with the storage manager subroutines

We call this set of subroutines the *primitive subroutines*.

The primitive subroutines are represented in the Smalltalk Virtual Image as methods with a special flag that says to run the corresponding subroutine rather than the Smalltalk-80 bytecodes. When the interpreter is executing the code to send a message and finds one of these flags set, it calls the subroutine and uses the value returned from it as the value of the method. The number of these methods in Smalltalk-80 is small (around one hundred) in order to keep the rest of the system as flexible and extensible as possible. We will not list those methods that are primitives, but will refer the reader to *Smalltalk: the Language and Its Implementation* (Goldberg, Robson, and Ingalls, 1981) for details.

A few of these primitive methods are executed so often that even the cost of looking them up in their classes' method dictionaries would be excessive. These methods are instead represented as special versions of the Send Message type of bytecodes. The message + , for example, is represented this way. When this bytecode is executed and the top two elements of the stack are small integers, then the primitive method is called as a subroutine. When this bytecode is executed and the top two elements of the stack are not small integers, then the + message is sent normally.

Conclusion

The Smalltalk-80 Virtual Machine is a fairly small computer program that consists of a storage manager, an interpreter, and a set of primitive subroutines. The task of implementing a Smalltalk-80 Virtual Machine for a new target computer is not large (especially when compared with the task of implementing other large programming systems) because most of the functions that must usually be implemented in machine code are already part of the Smalltalk-80 Virtual Image that runs on top of the Virtual Machine.

The Smalltalk-80 Virtual Machine could also be implemented in hardware, although this has not yet been done. Such an implementation would sacrifice some of the flexibility of software, but it would result in the performance benefits that hardware provides. Given the evolving nature of Smalltalk, it may not yet be time to implement the Virtual Machine in hardware: new Smalltalks that are more powerful would likely need at least small changes in Virtual Machine definition and implementation. However, hardware assists to Smalltalk-80 Virtual Machine software can greatly improve performance. Writable microcode stores for the pieces of code that are frequently run, hardware assists for graphics, or hardware assists for the fetching of bytecodes could all potentially improve the performance of a Smalltalk-80 Virtual Machine implementation. ■