Moving Structures between Smalltalk Images

Steven R. Vegdahl

Computer Research Laboratory
Tektronix Laboratories
P.O. Box 500, M.S. 50-662
Beaverton, OR 97077
(503) 627-6010

Abstract

There are a number of reasons why a user might want to move data structures between Smalltalk images. Unfortunately, the facilities for doing this in the standard Smalltalk image are inadequate: they do not handle circular structures properly, for example. We have implemented a collection of Smalltalk methods that handles circular structures; in addition, these methods have a number of other advantages over those provided in the standard image. This paper is largely a discussion of the issues that arose during their design, implementation, and use.

1. Introduction

There are two types of data with which a Smalltalk programmer may deal. The first are *Smalltalk objects*, which reside in an image; the second are *external data*, which can also reside outside an image—in a text file, for example. From Smalltalk's viewpoint, an object has a rich semantic structure that includes a class, instance variables, and associated methods. External data, on the other hand, is essentially just a sequence of uninterpreted bits. Because communication between Smalltalk images occurs only through external data, the movement of an object from one image to another requires that it be transformed into an external format—such as a character string—by the *source image*, and then reconstructed by the *target image*.

In the standard Smalltalk image, a mechanism—namely, the *storeOn:* and *readFrom:* methods—is provided for writing and reading object definitions to and from text files [Goldberg 83]. Unfortunately, this mechanism does not handle circular structures. Another problem is that this facility formats the text in compiler-compatible format; recreating the object is done by compiling and executing the text. In many Smalltalk systems—including ones on which the author has worked—there is an upper limit on the number of literals that may be included in the text of a method. The standard Smalltalk facility for reading and writing objects, then, tends to work only for small, non-circular objects.

This paper discusses some of the problems that arise when one considers moving objects between images in a more general manner. We have implemented a collection of methods that allows most structures to be written out; there are still cases, however, that cause problems. This paper is largely based on our experience in writing the system. It is assumed throughout that the reader is fluent in Smalltalk.

2. Implementing structure-reading and -writing methods

The motivation for implementing the methods for structure-reading and -writing was that the author had some rather complex data structures to distribute to several co-workers. Because the data structure required a relatively long time—and a number of auxiliary classes and methods—to generate, the task of creating it from scratch on each image was quite cumbersome. Instead, he implemented some rather simple methods for reading and writing circular structures; these methods have since been enhanced, but some problems remain.

The *storeStructureOn:* method recursively writes a representation of an object onto a *Stream*, keeping a mapping of objects to unique integers. The first time an object is seen in the depth-first traversal, its structure is written out; when an object is encountered that has already been seen, its corresponding integer is written rather than the entire object definition. Writing out an object in this manner consists of writing its class name and size, followed by the definitions of all its variables, which are accessed using the *basicAt:* and *instVarAt:* selectors.

The *readStructureFrom:* method creates an object in a similar way. It recursively parses the input text, using *basicAt:put:* and *instVarAt:put:* to load each object's values.

The implementation described so far is quite straightforward. Recursive methods can easily be implemented for *Object* that transform object-structures to and from character strings. Such an implementation, however, is not adequate. Certain Smalltalk classes, such as *Symbol* and *SmallInteger*, must be treated specially. Additional issues arise when one considers moving objects between images that run under different interpreters—ones with different word-sizes and floating-point precisions, for example. Still another set of issues must be faced when considering images whose class-definitions are not identical.

The internal text format is readable by humans only with great difficulty. This is not surprising, as human readability was not a goal of this project. Results by Lamb [Lamb 83] suggest that this need not be the case.

3. Mapping unique objects

There are several reasons why simply writing and reading the structures recursively while keeping track of previously seen objects does not work. One of the most obvious is that certain objects must be unique in any image. Among such objects are *nil*, *true*, *false*, and objects of class *SmallInteger*, *Symbol*, *Character*, *Class* and *MetaClass*; other global variables, such as the dependency structure, should also be unique. When such an object is read into the target image, a new instance should not be created; rather, the existing object in the target image should be used.

In our implementation, objects that are a kind of *Number* are written and read using *storeOn:* and *readFrom:*. This is safe, because numbers do not contain object pointers. (If a subclass of *Number* were created whose instances contained object pointers, the methods that assume pointer-freeness would have to be overridden.) Because two interpreters may differ in the range of integers that are representable as *SmallIntegers*, this provides some amount of interpreter independence. The effect of using using this notation is invisible to the user, except that two distinct, but equal, large integers might be mapped to identical objects. Because we do not expect that Smalltalk programmers depend on the internal structure of numbers, we do not anticipate this being a problem.

Several other minor modifications were made so that *Numbers* could be read properly. First, objects of class *Float* were written out to 9 decimal digits of accuracy rather than the default 6. Before this was done, the value of a *Float* object could change when written out and read back into the same image. Although a significance of 9 digits is sufficient for our Smalltalk interpreters, it is not necessarily sufficient for all interpreters. Mapping floating-point values between architectures is a difficult problem in general [Lamb 83]. Another minor modification was necessary when reading in a *Fraction* value because *Number-readFrom:* would stop when it found the numerator, claiming rightfully to have a (*Integer*) number.

The class *String* also had its storing method modified to use the standard Smalltalk *printOn:* method, rather than listing out each character in verbose format. Thus, the two-character string containing the characters 'h' and 'i' is stored as 'hi' instead of something like 'Character(104) Character(105)'. The motivation for making this modification was conciseness, although it also improves human readability.

There are a number of objects—*nil*, *true*, *false*, *Symbols*, and *Classes*, for example—for which there is an obvious object (either existing or easily-creatable) to which *readStructureFrom:* can map. Dealing with global variables, on the other hand, is more difficult. Currently, global variables in the source image are mapped into the correspondingly-named objects in the target image. The rationale behind this is that what the user usually wants is not a new copy of a global variable—such as the system

dictionary or the active process list—but rather the corresponding global object in the target image. Problems can arise when this strategy is used, however, due to the user's ability to declare an arbitrary global name to have an arbitrary value:

- A global name that is declared in the source image may not be defined in the target image. In the present system, an error message is presented to the user if such a name is not defined in the target image.
- A global in the source image may have, say, an integer value, such as the number 2. Having all references to the integer 2 in the source image correspond to some other value in the target image does not seem to be a good idea. In the present system, "simple" values are exceptions to the rule.
- Two globals in the source image may refer to the same object, resulting in an ambiguity as to which name should be written out. In the present system, the decision depends upon the order in which they appear in the internal representation of the system dictionary, and is thus somewhat arbitrary. This is clearly not an ideal solution.

Another issue that involves global variables is that of the Smalltalk dependency structure. Our implementation makes no attempt to represent dependencies between objects. In some cases, this will be what the user wants; in some cases it will not.

4. Mapping abstract objects

There are some classes for which the physical structure is not the correct thing to map. In such cases, it is desirable to represent a more abstract notion of an object and its contents. This is the case for the class *Set* and its subclasses. If the physical structure is simply mapped to a target image, the resulting structure may be not be consistent because hash values are not necessarily preserved across images. Thus for objects of class *Set*, the (abstract) contents—rather than the underlying structure—are written out. To read in a *Set* object, first an empty set is created; it is then filled by successively adding each element as it is read in. An alternative—and likely more efficient—implementation would be to read the structure into the image in the standard manner, and then to send the *rehash* message to the object upon completion.

Actually, neither of the above strategies handles all cases correctly. Consider an *Array* object *X* that contains a *Set* object, *S*, as its first element. Let us furthermore assume that *S* contains *X* (see Figure 1). If the *read* method is used to read in *X*, it is likely that *S* would be incorrectly hashed upon completion. The reason for this is that the hash value of *X* depends on the hash values of its first and last elements, but the last element of *X* is *nil*, not yet having been loaded at the time *S* is read in (and rehashed). The sequence of operations would be something like:

- A skeleton version of the array *X* is created; that is to say, it is an object whose size and class are correct, but whose elements are all *nil*.
- The set *S* is created, filled and rehashed. *S* now contains *X*, but *X* is still an array of *nil*.
- The elements of *X* are filled, thereby implicitly changing its hash value. The reading in of set *S*, however, has already been completed; it is therefore not rehashed.

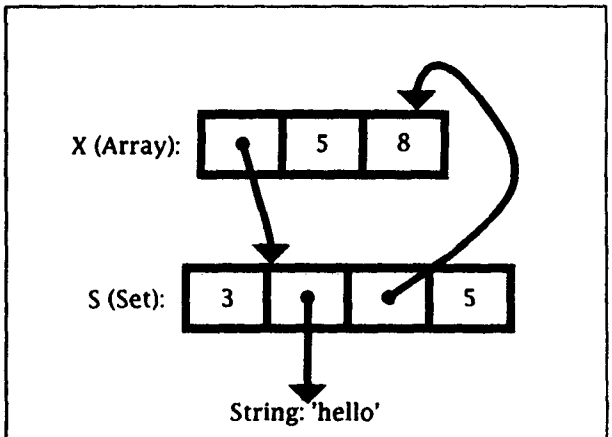

Figure 1. Structure that fails to hash properly.

One could argue that hashing objects that have circularities is a poor idea, and that the user should not expect it to work; for the above example, the author might agree. On the other hand, the pointer to *S* in the example need not be a *direct* reference; it is possible for an arbitrary indirection to cause such improper hashing to occur. To solve this problem in general, it is probably necessary to keep a list of *Sets* created as the structure is read in, and to send each the *rehash* message each after all objects have been created.

Another class of abstract object that causes problems is that of *CompiledMethod*. Our initial implementation reported an error if an attempt was made to write out a compiled method. This had the unfortunate implication that *SortedCollections* could not be written out using our methods, because a *SortedCollection* contains a *BlockContext* as an instance variable, which in turn (indirectly) refers to a *CompiledMethod*.

Our first attempt at storing compiled methods was to write out the header fields, object pointers, and byte codes after decoding them from their rather cryptic format. As long as the use of the object was confined to images running the same interpreter, this worked well; problems occurred when a second (32- rather than 16-bit) interpreter was made available. The format of a *CompiledMethod* was completely different under the 32-bit interpreter, consisting of three objects rather than one. Furthermore, the bytecode positions within the object differed, even though the bytecodes themselves were the same; any integer variable that referred to a bytecode position in the source image, had to be modified when mapped to the target image.

The current implementation routinely moves compiled methods between the two image-types at Tektronix. To solve the problem in a general way—for use with systems that compile to native code rather than using bytecodes, for example—it appears to be necessary to decompile the method into a source-like text, and to represent instance variables that are used as program counters—such as program counter values in *BlockContexts*—in a source-relative form.

The subject of *BlockContexts* brings to mind another potential problem. If an attempt is made to write out an active block context or method context, the entire call chain would be written out because each context contains a pointer to its sender. Because we believe that this is rarely what the user wants, such fields are always written out as *nil*.

5. Mapping redefined objects

Still another problem encountered in attempting to solve the structure-writing problem was that of moving objects between images whose class definitions are not consistent. The current implementation employs a simple—and highly inadequate—strategy: If the number of instance variables for a class is the same in both images, it maps them by position; otherwise it reports an error. The user is responsible for ensuring that the definitions are equivalent.

A simple way of improving this strategy would be to write out an ordered list containing the names of all instance variables for each class in the source image. (Classes that had no instances written out, of course could be excluded.) Upon reading the object into the target image, then, the names could be checked, reporting an error if there is a discrepancy. A slightly more robust technique would be that of checking whether the instance variables differed only in that they were permuted, and then applying the appropriate transformation to the objects in the target image.

More elaborate techniques for solving the problem are also possible. One would be to allow the user to specify a message-name that is to be sent to each newly-created object. The purpose of such a method would be to fill in the method's instance variables in a manner prescribed by the user.

These issues are similar to those of the *mutation* that takes place within an image when the definition of a class changes. What typically happens during mutation is that each instance (object) of the modified class is changed so that it has the correct number of instance variables; new instance variables are initialized to *nil*; also, corrections are made for any instance variables that may have changed position.

Another consequence of moving inconsistently-defined objects between images is that moving compiled methods between images must be done at the source-level. Machine- or byte-code instructions refer to instance variables by position, not by name, resulting in incorrect execution if instance-variable positions have changed.

6. Conclusions

We have implemented a mechanism that is a first attempt at allowing general structures to be moved between Smalltalk images. It makes a reasonable attempt at doing the "right thing" in most cases; in others, it does the "easy" (as well as wrong) thing. The system, which consists of about 20,000 bytes of source code, has become the method of choice within our user community for moving objects between images, including those that run under different interpreters. Aside from not being able to handle "unusual" structures, it has been quite successful.

The moving of arbitrary structures between Smalltalk images—even ones that run under the same interpreter—is not a trivial task. Problems and ambiguities arise involving global variable names, hash values, and inconsistent class definitions. Still more problems arise when an attempt is made to move structures between images that run on different interpreters; many of these are similar to those faced when *cloning* an image so that it can run on a new interpreter. These include compatibility of floating-point formats, and the representation of *CompiledMethods*.

Some of the problems encountered—such as the interaction between circularity and hashing, and the representation of *CompiledMethods*—appear to have solutions. Many of the of the other problems appear either to be intrinsically difficult, or to require human intervention. These include the preservation of distinct (but equal) large integers, mapping global variables, dealing with inconsistent class definitions, and the taking of a reasonable action in mapping dependencies. Although solving the problem in a general fashion does not appear to be possible, it is our experience that a tool which solves even a portion of the problem is quite valuable.

References

[Goldberg 83] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Massachusetts (1983).

[Lamb 83] D.A. Lamb, *Sharing Intermediate Representations: The Interface Description Language*, PhD thesis, Carnegie-Mellon University (1983).

Appendix A. Default structure-writing code.

The following are three methods defined in *Object* that implement the core of the structure-writing mechanism. Classes for which these methods are inappropriate must override them. The *StructOutputTable* object is an auxiliary object that keeps track of objects seen so far, as well as the image's globals.

```
storeStructureOn: aStream
  "Writes on aStream a circular description of the
   receiver."

  self
    storeStructureOn: aStream
    auxTable: (StructOutputTable new).


storeStructureOn: aStream auxTable: auxTable
  "Stores the definition of an object onto aStream,
   given that the objects contained in auxTable
   have already been seen. This method is rarely
   overridden. In our implementation, it is only
   overridden by Number. In the normal case, the
   object's id number is written out, followed by
   a '#' and its class name, and finally a set of
   parentheses enclosing its definition."

  | expand |

  expand ← false.
  aStream
    nextPutAll:
      (auxTable
        idOfElement: self
        ifNew: [:num | expand ← true]).
    printString.
  expand
    ifTrue:
      [self isUniqueValue
        ifFalse:
          [auxTable
            if: self
            isGlobal: [:name |
              aStream nextPutAll: '&', name. ^self]].
      aStream
        nextPutAll: '#', self class printString, '('.
      self
        storeDefinitionOn: aStream
        auxTable: auxTable.
      aStream nextPutAll: ')'].
```

```
storeDefinitionOn: aStream auxTable: auxTable
 "Stores all relevant information about the
  contents of an object. First the number
  of slots in its variable part is written
  out (or '-' if the object has no variable
  part. Then the object's instance
  variables are written, followed by its
  array-variables."

 aStream
  nextPutAll:
  (self class isVariable
   ifTrue: [self basicSize printString]
   ifFalse: ['-']).
 1 to: self class instSize do: [:i |
  aStream nextPutAll: ' '.
  (self instVarAt: i)
   storeStructureOn: aStream
   auxTable: auxTable].
 1 to: self basicSize do: [:i |
  aStream nextPutAll: ' '.
  (self basicAt: i)
   storeStructureOn: aStream
   auxTable: auxTable].
```

Appendix B. Example structure-writing output.

The following are examples of the output from *storeStructureOn:*. The first example (see Figure 2) was created by evaluating the Smalltalk expression

```
Array
 with: nil
 with: 5
 with: 'Hello Charlie'
 with: nil
```

in a workspace. This array, whose four elements are *nil*, the *Integer* 5, the *String* 'Hello Charlie', and *nil*, respectively, was written by our system as

```
1#Array(4 2#UndefinedObject(-) 3%5
 4#String('Hello Charlie') 2)
```

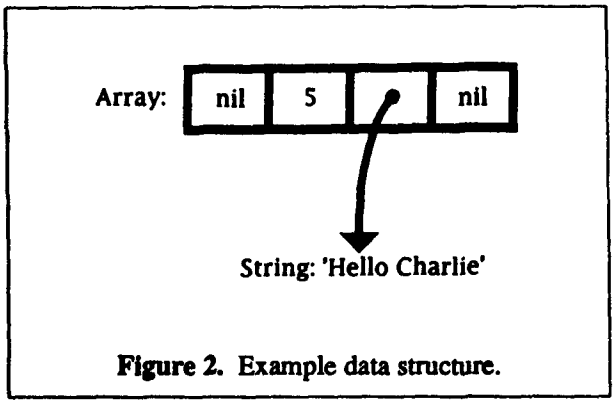

Figure 2. Example data structure.

The second example (see Figure 3) was produced by executing

```
s ← Set new.
o ← Array with: s with: 3/4 with: Form.
a ← Array with: #xyz with: o.
s add: #xyz; add: Form; add: a.
```

in a workspace. *Set s*, contains the class object *Form*, the *Array a*, and the symbol *#xyz*. *Array a*, in turn, contains the symbol *#xyz*, and the *Array o*, which contains the *Set s*, the *Fraction* 3/4, and the *Class* object *Form*. The resulting output from our system when applied to *s* was:

```
1#Set(2&Form 3#Array(2 4#Symbol('xyz')
 5#Array(3 1 6%(3/4) 2)) 4 )
```

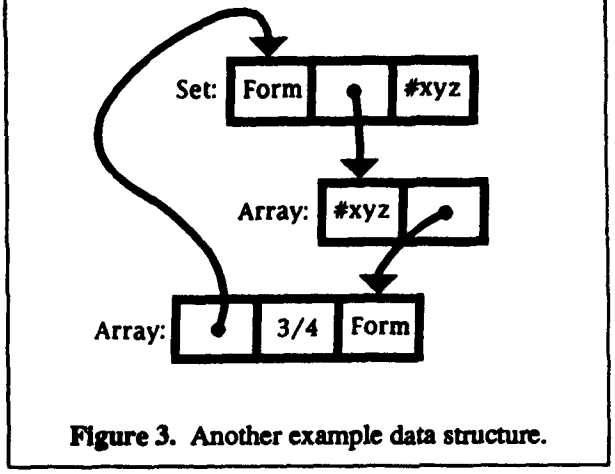

Figure 3. Another example data structure.