Extending the Squeak Virtual Machine

Andrew C. Greenberg

NetWolves Technology Corporation, Inc.

Why Extend Squeak?

Extending Squeak's virtual machine assumes an answer to an important threshold question: Why do it? Squeak is an exquisite and highly portable programming environment that is capable of performing most things a programmer might want to do. Squeak has robust memory management and debugging tools that protect the system from catastrophic errors and inform programmers as to their cause. Squeak supports Smalltalk, and therefore an object-oriented programming paradigm that results in highly reusable and extendable code. Squeak programs are remarkable in their capacity to perform complex applications across many platforms – identically pixel-for-pixel.

In contrast, each virtual machine ("VM") extension bypasses the Squeak object memory model. Therefore, the extension reintroduces the possibility of hardware crashes due to memory leaks, hanging pointers, address errors and the panoply of things that can go wrong in traditional programming. VM extensions often rely on fixed assumptions about the internal structure of the objects on which they operate, and thus limit the scope of reusability and extensibility of objects that depend on the extension. Otherwise routine operations on Smalltalk objects so extended can result in strange and unpredictable behaviors. Finally, VM extensions tend to introduce machine dependencies.

Notwithstanding all of this, however, there are sometimes compelling reasons to extend the Squeak VM. Among these are to achieve improvements in performance and to obtain enhanced functionality not otherwise possible without an extension.

Squeaking out more speed

Squeak programs are not compiled to native machine language, but are compiled to an intermediate representation called bytecodes. Bytecodes are interpreted by the Squeak VM, a computer program that is, in turn, executed on the target machine. Because of this intermediate processing, the overall system typically runs slower, sometimes much slower, than a corresponding C language program that was directly compiled to machine language. Additionally, the Squeak object model carries an unavoidable overhead, since virtually every program operation requires some machine resources to support the Smalltalk message sending and memory management semantics.

In most cases, benefits in portability, power and safety that derives from the Squeak design decisions may substantially outweigh any cost in speed and other resources. Moreover, as costs plummet for increasingly high-speed processors, demand for "on the metal" speed diminishes. Nevertheless, there are times when the need for speed is paramount – where the lack of speed equates to a lack of functionality.

Modern software design methodologies focus on postponing attention to performance issues until later stages of the system life cycle: "Make it work, make it right, then make it fast." Once a working and sound model of a system exists, traditional "tuning" solutions entirely within Smalltalk often provide adequate performance solutions. Since programs tend to spend the vast majority of their time executing tiny portions of its program code, this approach can often yield excellent and efficient results while permitting a programmer to focus energies on refining only small portions of the code. Using a Squeak profiler to identify program bottlenecks, particular method or methods can often be identified and improved to a degree sufficient to satisfy a program's requirements.

This approach works well particularly in view of Smalltalk's object-oriented infrastructure. A complete program can often be improved enormously without making changes to the key program logic merely by changing the internal representation and implementation of a single class. Where bottlenecks are not so localized, a program can often be refactored to facilitate speedup efforts. In other cases, straightforward speedups can be obtained by specializing generalized system classes used by the program. For example, a program using the Smalltalk Set classes can often be improved by substituting (without changing the main code) another class that efficiently exploits particular properties of the underlying data.

Sometimes, however, traditional tuning fails to provide speed-up sufficient for an essential function. In such cases, replacing a performance bottleneck with straight-line machine language may make possible functionality that could not be adequately delivered otherwise.

Roaring like a mouse: new power and functionality.

Squeak is extraordinary. The standard image contains arbitrary precision arithmetic, a comprehensive set of collection protocols, several graphic user interface frameworks, a flexible 3D graphics engine, a reusable web server, support for most traditional internet protocols and powerful sound processing capabilities to name just a few of the key built-in "goodies." Underneath, Squeak provides a vast set of primitives providing low-level access to many important machine functions.

Nevertheless, the functional capability of computers continues to increase, and Squeak's designers could not have anticipated every possible need for a system-level primitive. Sometimes, the requirements for access to low-level drivers or local operating system functionality is essential to perform specific functions. Without the ability to extend Squeak, some hardware might not be usable at all.

In other cases, an existing machine-specific or portable non-Smalltalk library may already exist, providing specific capabilities that would be necessary or helpful for a Squeak application. Particularly where the library is of general utility and effectively maintained by others, it may be desirable to code an extension for Squeak to provide direct access to the functionality of that library, rather than diverting resources unnecessarily to re-invent the wheel in Smalltalk.

These two issues, access to low-level drivers and access to already existing non-Smalltalk libraries, are among the most common reasons for using Squeak extensions.

Anatomy of an Extension

Squeak can be extended in various ways: by rewriting the interpreter, by adding a numbered primitive or by adding a named primitive. The first two require rebuilding the entire Squeak system. By far, the more flexible and adaptable solution –and the primary subject of this chapter – are named, or "pluggable" primitives. On most systems, these named primitives can be implemented as shared libraries, such as Windows DLL libraries.

Once a plugin is made available to the VM using native operating system conventions (in the MacOS, for example, by dragging plugin files into certain folders), a Smalltalk method can call one of its primitives using a specialized extension of the Smalltalk syntax for primitives. This method (the *calling method*) and the primitive functions in the plugin communicate in a specialized manner. Ultimately, the primitive returns control to Squeak with a Smalltalk object as its answer.

The process of extending Squeak with named primitives entails the following:

1. Creating a Smalltalk interface to the named primitives; and
2. Creating the Smalltalk plugin and its named primitives.

The Plugin Module

The plugin itself is an external library of machine-language routines, including (i) one or more parameterless functions (the primitive functions) each returning a 32-bit integer result, and (ii) a separate function taking a single 32-bit parameter, which must be named **setInterpreter**. The primitive functions may be generated in any manner convenient to the author, but are typically generated from source code written in a subset of Smalltalk called *Slang*, which is then translated to C and, in turn, compiled to machine language.

The Interpreter Proxy

Prior to the first call of any primitive in the plugin, the Squeak VM calls setInterpreter, passing to the plugin a pointer to a data structure known as the *interpreter proxy*. The interpreter proxy includes pointers to various internal data structures of the VM and to a subset of the VM's internal subroutines. The plugin saves the interpreter proxy in shared memory for use by the primitives when communicating with Squeak.

Linkage Conventions for Primitive Functions

In operation, the Squeak VM represents every Smalltalk object as a 32-bit integer value, called an *oop.* To call a primitive, the VM creates a stack comprising oops representing the actual parameters and an oop representing the object to which the primitive message was sent. The primitive will thereafter either succeed or fail, and the fact of success or failure is recorded in a global VM variable named **successFlag**. If the primitive succeeds, the primitive is expected to pop the parameter and sender oops from the stack, and to put (or leave) in its place a single oop representing the answer to the message. If the primitive fails, then the primitive is expected to leave the stack unchanged. Strict compliance with these linkage conventions is essential to the proper operation of a primitive function.

Key to building extensions, therefore, is an understanding how Squeak interacts with primitives, and vice-versa. To write a primitive, one must understand in some detail how Squeak represents Smalltalk objects, the workings of the Squeak memory model and how representations of information can be shared between the VM and the primitive.

The rest of this chapter will discuss the mechanics of building a plugin VM extension. We will begin with a brief introduction to the Slang Smalltalk subset and how to use the Squeak translators and interpreters. We will then detail how Squeak represents Smalltalk data objects and how to use the interpreter proxy to access and manipulate that information. We will then outline the mechanics of how a primitive function can use the interpreter proxy to interact with the Squeak interpreter. Finally, we conclude by putting these pieces together concretely with an example plugin for manipulating large blocks of structured text.

Speaking in Slang

Although named primitive functions may be written in any language capable of generating a shared library with the appropriate linkage conventions, most are written in a subset of Smalltalk known as Slang. Slang code can be written and tested in the Squeak development environment, and then translated into C for compilation to a plugin.

A First Plugin

We will begin with a simple example, building a trivial pluggable primitive that answers the **SmallInteger** instance 17. We begin by creating a subclass of **InterpreterPlugin**.

```
InterpreterPlugin subclass: #ExamplePlugin
        InstanceVariableNames: ‘’
        classVariableNames: ‘’
        poolDictionaries: ‘’
        category: ‘VMConstruction-Plugins’
```

The Primitive Function

InterpreterPlugin provides the essential functionality for constructing a plugin, including code to declare and initialize the interpreter proxy (which will be stored in an instance variable named **interpreterProxy**) and the infrastructure for translating the plugin to C. To create our primitive function, open a browser on **ExamplePlugin**, and add the following method

```
answerSeventeen
        “Extension to answer the SmallInteger seventeen”
        self export: true.
        interpreterProxy
                pop: 1
                thenPush: (interpreterProxy integerObjectOf: 17)
```

The first statement has no effect when interpreted in Smalltalk, but during translation, identifies the translated procedure as one with an external label in the resulting library. The second statement pops the Smalltalk context stack and then pushes an object handle (called an *oop*) representing the SmallInteger instance 17.

.

The Smalltalk Interface

Once coded, this method can be tested by executing the following in a workspace:

```
ExamplePlugin doPrimitive: ‘answerSeventeen’
```

to obtain the answer “17.” While hardly useful, this code nevertheless illustrates how a primitive might be built. Once the extension is built, the Smalltalk interface can be written. Primitives are conventionally invoked by methods written substantially as follows:

```
answerSeventeen
        “Answer the SmallInteger seventeen”
        <primitive: ‘answerSeventeen’ module: ‘ExamplePlugin’>
        ^ExamplePlugin doPrimitive: ‘answerSeventeen’
```

This method, when executed, will seek to invoke the answerSeventeen primitive, but when the primitive is not available or when the primitive is available but fails during execution, it begins to interpret the remaining Smalltalk code. Assuming that this method were added to class Foo, one would test the primitive by executing the following in a workspace:

```
Foo new answerSeventeen
```

Since we have not yet built the plugin module, the primitive will fail, and the result of the interpreted code will be answered.

Translating and Building the Plugin

Once you are satisfied that the Slang program is correct, a plugin library can be created by executing the following:

```
ExamplePlugin translate
```

to invoke the Slang translator and generate a file, “ExamplePlugin.c,” containing the C code corresponding to the Slang set forth in class ExamplePlugin. That C program may then be compiled using native tools to generate the plugin library.[1]

Slang: A Brief Summary

Expressions

Slang recognizes Smalltalk literals, including integers, characters, symbols and strings, which are translated into corresponding C-language constants. Symbols are translated into strings. Array literals are not permitted. Assignments from expressions to identifiers are translated into C-language assignments from the translation of the expression to the corresponding identifier.

Unary Message Sends

Unary message sends are generally translated into a procedure call of the procedure identified, passing the receiver as a parameter. Thus:

```
anObject frobosinate
```

is translated to

```
frobosinate(anObject);
```

Binary Message Sends

Binary message sends are generally translated into a procedure call of the procedure identified, passing receiver as a first parameter, and the rightmost argument as a second parameter. Thus:

```
anObject frobosinateWith: aWidget
```

is translated to

```
frobosinateWith(anObject, aWidget);
```

Keyword Message Sends

Keyword message sends are generally translated into a procedure call of the procedure identified (by conacatenating all of the keywords without colons), passing the receiver as a first parameter, and the remaining arguments in order. Thus:

```
anObject frobosinateWith: aWidget andWith: anotherWidget
```

is translated to

```
frobosinateWithandWith(anObject, aWidget, anotherWidget);
```

[1] Translated C code may depend upon one or more include files, copies of which are stored in the standard Smalltalk image. You may generate these in text file form by executing the expression, “InterpreterSupportCode writePluginSupportFiles,” in a workspace.

Message Sends to Self or To InterpreterProxy

An exception to the general rules stated above occurs when messages are sent to either of the “special objects” self or interpreterProxy. Messages sent to *self* are translated as above, but without self as the initial parameter. Messages sent to interpreterProxy are also translated as above, except that the function call will have the string “interpreterProxy->” prepended. Thus:

```
self frobosinateWith: a
```

is translated to

```
frobosinateWith(a);
```

and

```
interpreterProxy integerObjectOf: 17
```

is translated to

```
interpreterProxy->integerObjectOf(17);
```

Builtin Message Sends

Certain messages are not translated in accordance with the preceding rules, but rather to C-language expressions with semantics similar to the corresponding Smalltalk operation:

```
&, |, and:, or:, not, +, -, *, /, //, \\, <<, >>, min:, max:,
bitAnd:, bitOr:, bitXor:, bitShift:, bitInvert32, raisedTo:
<, <=, =, >, >=, ~=, ==,~~
```

Other messages are also given special meanings, as described below.

Arrays

The expression

```
foo at: exp
```

is translated to

```
foo[exp]
```

and the expression

```
foo at: exp1 put: exp2
```

is translated to

```
foo[exp1] = exp2
```

The messages **basicAt:** and **basicAt:put:** are translated in the same way.

Control Structures

The following Smalltalk “control structure messages” are translated to C-language statements with similar semantics:

```
[stmtList] whileTrue: [stmtList2]
[stmtList] whileFalse: [stmtList2]
[stmtList] whileTrue
[stmtList] whileFalse
exp1 to: exp2: do: [stmtList]
exp1 to: exp2: by: exp3 do: [stmtList]
exp1 ifTrue: [stmtList]
exp1 ifFalse: [stmtList]
exp1 ifTrue: [stmtList] ifFalse: [stmtList]
exp1 ifFalse: [stmtList] ifTrue: [stmtList]
stmt1. stmt2
```

Note that the square brackets are used here for syntactic purposes only in these control structures. Slang does not support Smalltalk code blocks.

Methods

A method in Smalltalk will be translated to a C language function returning an integer. If the method is declared in keyword form, the C language function will be named by concatenating all of the keywords, but without semicolons. Methods may be translated to be called directly as primitives from Smalltalk, as described above, or may be translated to be called as subroutines from C language code, as described below. Temporary variables will be translated as local variables for the function. Thus:

```
frobosinateWith: a andWith: b
        | temp |
        . . .Slang code . . .
```

will be translated as

```
int frobosinateWithandWith(int a, int b)
{
        int temp;
        . . . Translated C code …
}
```

If you should want the procedure to return a value with a type other than int, you may use the directive

```
self returnTypeC: ‘char *’
```

C Language Declarations and Coercion of Expressions

Unless you specify otherwise, Slang will translate all method temporaries into C variables defined as integer types. You can declare the variables differently in translated code by using the Slang directive:

```
self var: #sPointer declareC: ‘char *sPointer’.
```

To satisfy C type checking semantics, it may be necessary to direct Slang to coerce an expression from one type to another. This can be accomplished with the Slang directive:

```
self cCoerce: aSmalltalkExpression to: ‘int’
```

Suppose the object at the top of the Smalltalk stack corresponded to an Array of characters such as a Smalltalk instance of class **String**. You might use the following Slang code to access its elements:

```
self var: #stringObj declareC: ‘int stringObj’.
self var: #stringPtr declareC: ‘char *stringPtr’.
self var: #stringSize declareC: ‘int stringSize’.
. . .
stringObj ← interpreterProxy stackValue: 0.
stringSize ← interpreterProxy stSizeOf: stringObj.
stringPtr ← self
        cCoerce: (interpreterProxy arrayValueOf: stringObj)
        to: ‘char *’.
```

Global Variables

Global variables for a plugin are declared in Smalltalk as instance variables of the **InterpreterPlugin** subclass, and can be further declared for purposes of C-language type definitions by adding a method named **declareCVarsIn:** to the *class* side of the **InterpreterPlugin** subclass. With global declarations, however, a string, rather than a symbol is used as the parameter for the **var:** keyword. An example follows:

```
declareCVarsIn: cg
        cg var: 'm23ResultX' declareC:'double m23ResultX'.
        cg var: 'm23ResultY' declareC:'double m23ResultY'.
        cg var: 'm23ArgX' declareC:'double m23ArgX'.
        cg var: 'm23ArgY' declareC:'double m23ArgY'.
```

Subroutines

Subroutines are useful in most programming situations. Writing named primitives is no different. For simplicity, many named primitive functions serve only as the “glue” between Smalltalk and the operative subroutines, merely reading and verifying the parameters from the stack and data from the receiver. That information is then passed to the operative subroutine, and returned to the glue primitive function, which clears and then pushes the return value back onto the stack in accordance with the linkage conventions.

You may call a Slang subroutine simply by sending the subroutine as a message to the “special object” **self**, for example:

```
self subroutineOn: aFirstValue and: aSecondValue
```

and in turn, the subroutine might be coded as follows:

```
subroutineOn: aFirstParm and: aSecondParm
```

```
self var: #aFirstParm declareC: ‘char *’.
self var: #aSecondParm declareC: ‘float’.
self returnTypeC: ‘float’.
. . . Slang code for the subroutine  . . .
```

No special linkage conventions need be followed for these internal Slang subroutines. Those linkage conventions apply only to procedures that will be directly called from Smalltalk using the “<primitive: . . . module:> mechanism.

The Slang-to-C translator automatically inlines a subroutine if the subroutine is either sufficiently short, or if it contains at or near the beginning of its code a Slang **inline:** directive of the form:

```
self inline: true.
```

Certain subroutines, for example subroutines containing the cCode: directive, will not be inlined even if the subroutine contains an **inline:** directive

Inline C-Language Code

Any C-language expression may be inserted into translated code with the expression

```
self cCode: ‘InternalMemorySystemSubroutine(foo)’
```

Of course, this code will not be executed in any way when the code is being interpreted. However, when translated to C, the string will be inserted verbatim, followed by a semicolon.

It is sometimes helpful, particularly when testing, to have certain Smalltalk code executed during translation. For that reason, the expression

```
self cCode: ‘InternalMemorySystemsubroutine(foo)’
        inSmalltalk: […Smalltalk code. . .]
```

will be translated as above, but when it is interpreted, the corresponding Smalltalk code will be executed.

TestInterpreterPlugin

Squeak also provides a second, upwardly compatible, Slang subset translator, the TestCodeGenerator. TestCodeGenerator is anticipated, in time, to supplant the present Slang interpreter. The TestCodeGenerator provides the same facilities as the translator described above, but also provides means for automatically generating plugin linkage code, and to greatly facilitate the writing of pluggable primitives.

To use the TestCodeGenerator, define the plugin class as a subclass of **TestInterpreterPlugin** instead of **InterpreterPlugin**. A complete description of TestCodeGenerator is beyond the scope of this article. Although documentation is regrettably quite scant as this article is written, more will be forthcoming in time. In the meanwhile, the following page from the Squeak Swiki may be helpful:

http://minnow.cc.gatech.edu/squeak/850

The next section discusses in some depth the internal structure of objects as they are represented in the Squeak Virtual Machine.

The Shape of a Smalltalk Object

In Smalltalk, everything is an object. Ultimately, however, objects must be represented in hardware as bits and bytes. Unaided by the abstractions of the Smalltalk model, primitives and the Squeak VM must concern itself with this uncomfortable reality. To write primitives, a programmer must know that the Squeak VM represents each and every Smalltalk object with a 32-bit integer value, known as an oop. The internal representations and interpretations of these oops, however, vary substantially depending upon the object represented by the oop. It may be helpful to consider four distinct categories, or "shapes," of Squeak object representations, and how they may be manipulated in Slang and C.

1. Smalltalk **SmallInteger** objects;
2. Other non-indexable Smalltalk objects;
3. Smalltalk objects indexable to oops;
4. Smalltalk objects indexable to data other than oops.

From a Smalltalk point of view, the first category is self-explanatory. The second category includes, for example, instances of classes **Boolean, Fraction** and **Object,** but also includes objects that contain indexed objects as instance variables, but are not themselves indexed, such as instances of classes **Form** and **Set.** The third category, bjects indexable to oops, include instances of class **Array**. The last, objects indexable to things other than oops, include objects indexable to byte or word data, such as instances of classes **LargePositiveInteger, ByteArray** and **WordArray.**

SmallInteger Objects

Primarily for efficiency reasons, Squeak represents 31-bit signed integer values (Smalltalk **SmallInteger** objects) with an oop containing that integer data. Oops representing other types of data are pointers to an object "header" stored elsewhere in memory. Since all object headers begin on word boundaries, these pointer oops are even numbers. Squeak exploits this fact by representing **SmallInteger**s in odd-numbered oops. **SmallInteger** Oops are therefore stored in the following format:

Bit Index	31 30 29 . . . 3 2 1	0
Data	31-bit SmallInteger Data	1

While every **SmallInteger** can be represented by a int-type C variable, the converse is not true. C int variables containing values greater than or equal to 2^{30} or less than -2^{30} will not "fit" into the 31 bits available for **SmallInteger** data.

Converting between **SmallInteger** Oops and Values

The interpreter proxy provides two methods to help back and forth between the oop representation of a **SmallInteger** and the actual numeric value represented:

In Smalltalk:	In C:
interpreterProxy integerObjectOf: value	interpreterProxy->integerObjectOf(value)
interpreterProxy integerValueOf: oop	interpreterProxy->integerValueOf(oop)

If the argument of **integerValueOf:** is an oop that represents a **SmallInteger** object, the method will answer a signed C value corresponding to the **SmallInteger**. The answer will otherwise be undefined, even if the argument does represent an integer object, such as a LargePositiveInteger. Conversely, if an argument to **integerObjectOf** is within the range of values that can be represented by a **SmallInteger**, then the method will answer with the corresponding **SmallInteger** oop. If a SmallInteger cannot represent the value, then the result will be undefined.

Testing For **SmallInteger** Oops and Values

The **isIntegerValue:** method answers a Boolean value reflecting whether the argument can be converted into a **SmallInteger** object. That is, whether the value is within the **SmallInteger** range. Conversely, **isIntegerObject:** answers a Boolean value reflecting whether the argument, treated as an oop, represents a **SmallInteger**.

interpreterProxy isIntegerObject: oop	interpreterProxy->isIntegerObject(oop)

```
interpreterProxy isIntegerValue: value          interpreterProxy->isIntegerValue(value)
```

Validating Conversion Function

The interpreter proxy provides a method for validating and loading values that are expected to be **SmallInteger**s with a single call:

```
interpreterProxy                                interpreterProxy->
        checkedIntegerValue: oop                        checkedIntegerValue (oop)
```

This method will check whether the oop is a **SmallInteger** value, and if so, return the corresponding integer value. If the oop is not a **SmallInteger** value, then the successFlag is set to false, and the result will be undefined.

Long Integer Values Outside the **SmallInteger** Range

On occasion, it is useful to have an extension pass or return a full 32-bit integer value. The interpreter proxy provides functions for manipulating unsigned 32-bit values:

```
interpreterProxy                                interpreterProxy->
        positive32BitIntegerFor: value                  positive32BitIntegerFor(value)

interpreterProxy                                interpreterProxy->
        positive32BitValueOf: oop                       positive32BitValueOf(oop)
```

The **positive32BitValueOf:** method will accept either a **SmallInteger** or a 4-byte LargePositiveInteger object and answer a value that can be saved into and manipulated as an unsigned long integer. The **positive32BitIntegerFor:** method will convert a value stored in an unsigned long int C variable and return an oop for a corresponding **SmallInteger** if the value is within the **SmallInteger** range, or will return an oop for a LargePositiveInteger.[2]

Other Non-indexable Objects

All other oops are represented in the VM as pointers to internal data structures containing further descriptions of the oop. The majority of Smalltalk classes have this shape.

Objects Without Instance Variables

Some objects with non-indexable shape bear no other data than the class of which they are a member. The only thing that a named primitive can meaningfully do with the oop for such an object is to assign the value

[2] Should a LargePositiveInteger oop be created during a call to #positive32BitIntegerOf:, a garbage collection may occur, which in turn might invalidate oops stored in other C variables. See the section below on Memory Management and Garbage Collection.

to another oop instance “slot,” or to compare it with other oops. Oops for some important special objects are provided by the interpreter:

In Smalltalk:	In C:
interpreterProxy falseObject	interpreterProxy->falseObject()
interpreterProxy nilObject	interpreterProxy->nilObject()
interpreterProxy trueObject	interpreterProxy->nilObject()

It is important to note that since the oop for Smalltalk **true** is likely to be non-zero, it will test as true in a C-language if-statement. However, the same is also true for the oop representing the Smalltalk object **false**. Accordingly, to determine in C whether an oop represents Boolean truth, one should instead code something like the following:

```
if (booleanOop == interpreterProxy->trueObject()) {. . .}
```

As an alternative, Smalltalk Boolean objects can be converted to and from C-language Boolean values with the following operations:

interpreterProxy booleanValueOf: oop	interpreterProxy->booleanValueOf(oop)

The **booleanValueOf:** method, like **checkedIntegerValue:,** will first check that the oop represents an object of type Boolean, and will set the successFlag to false if it does not. If the oop is valid, then the method will answer a corresponding C-language Boolean value.

As with oops for **true** and **false**, the oop for Smalltalk **nil** will not bear the same integer value as the C language constant NULL. The following code:

```
if  (oop == NULL) {. . .}
```

will not behave as expected, because the result of the C expression is almost always likely to be false, regardless of whether oop actually represents the Smalltalk object **nil**. To test in C whether an oop represents nil, you should instead code something like the following:

```
if (oop == interpreterProxy->nilObject()) {. . . }
```

Objects With Named Instance Variables

The vast majority of Smalltalk Classes, however, define non-indexable objects with one or more instance variables. These instance variables are stored in numbered slots, beginning at 0. The slot numbers correspond to the sequence in which those names are listed in the Class definition. The interpreter proxy provides the following functions for manipulating oops corresponding to objects with such a shape:

In Smalltalk:	In C:
interpreterProxy	interpreterProxy->

```
        fetchWord: slot                          fetchWordofObject (slot,oop)
        ofObject: oop

interpreterProxy                         interpreterProxy->
        firstFixedField: oop                            firstFixedField(oop)
```

The value returned by **fetchWord:ofObject:** is an oop corresponding to the object stored in the corresponding instance variable. The value returned by **firstFixedField:** is a word-based pointer which can be indexed to return a corresponding instance variable (or in Slang, a Word-based CArrayAccesor). Thus,

```
        p ← interpreterProxy firstFixedField:oop.
        0 to: numInstVars do: [:i | self foo: (p at: i)]

        int *p;
        p = interpreterProxy->firstFixedField(oop);
        for (i=0; i<numInstVars; i++) {foo(p[i]);}
```

will perform the function foo on oops for each instance variable in the object represented by oop.

The interpreter proxy provides facilities for extracting the contents of an instance variable from an object, type checking and, in the case of floats and integers, converting the value at the same time.

```
interpreterProxy                  interpreterProxy->fetchIntegerofObject(
        fetchInteger: slot                slot,oop)
        ofObject: oop

interpreterProxy                  interpreterProxy->
        fetchPointer: slot                        fetchPointerofObject (slot, oop)
        ofObject: oop:
```

As side effects, **fetchInteger:ofObject:** will fail unless the oop corresponds to a **SmallInteger**, and **fetchPointer:ofObject:** will fail if the oop corresponds to a **SmallInteger**.

For example, suppose rcvr contained an oop for an instance of a class that defined as follows:

```
        Object subclass: #Example
                instanceVariableNames: 'instVar0 instVar1'
                classVariableNames: ''
                poolDictionaries: ''
                category: 'Squeak-Plugins'.
```

Then, if instVar0 and instVar1 contains a ByteArray object and a **SmallInteger** object, respectively, you can load the oop pointers to those objects from rcvr as follows:

```
        oop0 ← interpreterProxy fetchPointer: 0 ofObject: rcvr.
```

```
oop1 ← interpreterProxy fetchInteger: 1 ofObject: rcvr.
```

Finally, valid Smalltalk values can be stored in slots of an object using the interpreter proxy methods:

```
interpreterProxy                    interpreterProxy->
        storeInteger: slot                  storeIntegerofObjectwithValue
        ofObject: oop                               (slot, oop, integerValue)
        withValue: integerValue

interpreterProxy storePointer:      interpreterProxy->
    storePointer: slot                      storePointerofObjectwithValue
    ofObject: oop                           (slot, oop, nonIntegerOop)
    withValue: nonIntegerOop
```

With **storeInteger:ofObject:withValue:,** integerValue will be converted to a **SmallInteger** oop and stored in the specified instance variable (or fail if it cannot be converted). The other method does not convert the oop (but will fail if nonIntegerOop is for a **SmallInteger**), and stores the object in the specified instance variable.

Objects Indexable to Oops

In addition to named instance variables, objects may contain a variable, indexable number of objects. A principle example is, of course, Smalltalk class Array. Indexable objects may contain oops referring to other Smalltalk objects, or they may contain raw numerical data as bytes or words. Like non-indexable objects, objects that are indexable to Smalltalk objects (but not objects indexable to data!) may also have any number of named instance variables.

Extracting the Indexable Instance Variables

Given an oop, you can obtain the size of the corresponding object (the value the object would answer if sent the message **size**) using the following code:

```
interpreterProxy stSizeOf: varOop          interpreterProxy->stSizeOf(varOop)
```

or, if you are only interested in the number of bytes consumed by the variable portion of theobject, you may use:

```
interpreterProxy                    interpreterProxy->
        byteSizeOf: varOop                  byteSizeOf(varOop).
```

Given an oop, you can then obtain or change the value of the oop of its elements using:

```
interpreterProxy                    interpreterProxy->
        stObject:  varOop                   stObjectat(varOop, index)
        at: index
interpreterProxy                    interpreterProxy ->
        stObject: varOop                    stObjectatput(varOop,index,value)
```

```
at: index
put: value
```

You can obtain a C pointer to the first indexable oop stored in the variable portion of the object with the following:

```
p ← self                                p = (int *) interpreterProxy->
    cCoerce: (interpreterProxy              firstIndexableField(oop)
        firstIndexableField: oop)
    to: 'int *'
```

after which you can address (or change) individual oops by indexing the array. Whether in Smalltalk or in C, all variable indexed objects referenced through the interpreter proxy are 0-based. Since **firstIndexableField:** answers objects of type 'void *,' the result should be coerced to 'int *'.

Extracting the Named Instance Variables

Objects indexable to oops may have named instance variables in addition to the indexable instance variables. You can test for the presence of such instance variables and manipulate them just as you would for non-indexable instance variables.

Testing For Objects Indexable to Oops

You may test whether an object is indexable (variable) using:

```
interpreterProxy isIndexable: oop        interpreterProxy->isIndexable(oop)
```

which will return true if the oop corresponds to any variable object regardless of whether the object is indexable to oops, bytes or words, or by using:

```
isPointers: oop
```

which will return true if and only if the oop corresponds to a variable object indexable to oops.

An Example

The following code reverses, in situ, the order of the objects in the variable Smalltalk object represented by **oop**.

```
a ← self cCoerce: (interpreterProxy firstIndexableField: oop) to: 'int *'.
i ← 0. j ← (interpreterProxy stSizeOf: oop) - 1.
[i<j] whileTrueDo:
    [t ← a at: i. a at: i put: (a at:j). a at: j put: t. i ← i + 1. j ← j - 1].
```

Or in C:

```
a = (int *) interpreterProxy->firstIndexableField(oop);
i = 0; j = (interpreterProxy->stSizeOf(oop)) – 1;
while(i<j) {t = a[i]; a[i]=a[j]; a[j]=t; i++; j--;}
```

Objects Indexable to 1-Byte or 4-Byte Values

Smalltalk permits the creation of objects with indexable instance variables containing binary data, each containing either a single byte or 4-byte word, each referenced by indexing. Objects of this shape may not contain named instance variable, and accordingly the methods described in the section for named instance variables are inapplicable to such objects.

Unsurprisingly, Smalltalk internally represents the instance variable data as an array of bytes or words. As with objects indexable to oops, pointers to a variable byte or a variable word object's array can be obtained using **firstIndexableField:.** (In the case of variable byte objects, you should coerce the result to 'char *' instead of to 'int *'). Likewise, you can obtain the size of such an object using **stSizeOf:** (noting that for word-indexable objects, the message will answer the number of 4-byte words, while for byte-indexable objects, the method will answer the number of bytes in the object represented by the oop). If you desire the number of bytes in the object, regardless of whether it is byte- or word- indexable, you may use **byteSizeOf:** instead.

You can check the shape of an oop with the following functions:

```
interpreterProxy isBytes: oop             interpreterProxy->isBytes(oop)
interpreterProxy isWords: oop             interpreterProxy->isWords(oop)
interpreterProxy isWordsOrBytes: oop      interpreterProxy->isWordsOrBytes(oop)
```

Finally, you can combine validation and conversion byte or word objects in one step using:

```
interpreterProxy                          interpreterProxy->
        fetchArray: index                 fetchArrayofObject(index,oop)
        ofObject: oop
```

to extract an oop pointer from a named instance variable array, or

```
interpreterProxy arrayValueOf: oop        interpreterProxy->arrayValueOf(oop)
```

to extract an oop pointer from any oop. Both methods will fail if the specified oop is not either byte- or word- indexable, and will return a pointer otherwise.

Special case of a Float

Floating point values, though in Smalltalk treated as scalar values, are represented as a 64 bit value in the form of an indexable object comprising two 32-bit words. The interpreter proxy converts oops representing Float objects into C values of type double, and vice-versa, with methods analogous to those used for **SmallInteger**s.

```
interpreterProxy                          interpreterProxy->
        floatObjectOf: aFloat                     floatObjectOf(aFloat)
interpreterProxy                          interpreterProxy->floatValueOf(oop)
        floatValueOf: oop
interpreterProxy                          interpreterProxy->
        fetchFloat: fieldIndex                    fetchFloatofObject(fieldIndex,
```

ofObject: objectPointer	objectPointer)
interpreterProxy isFloatObject: oop	interpreterProxy->isFloatObject(oop)

The Anatomy of a Named Primitive

Primitives are ordinarily invoked in the course of evaluating a Smalltalk expression, when a receiver is sent a message that has been defined as follows:

```
primitiveAccessorNameWith: object1 then: object2 andThen: object3
        <primitive: 'primitiveName' module: 'ExtensionPluginName'>
        "…
        Smalltalk Code to be executed if the primitive fails or cannot be loaded
        …"
```

Of course, the name of the method and the number and names of its parameter may vary. When the message is sent, the oop for the receiver is pushed upon a stack, each parameter is evaluated and pushed onto the stack in the order they appear in the method's definition. The VM global successFlag is set to true. The stack then looks substantially as follows:

(top) 0	oop for object3
1	oop for object2
2	oop for object1
(bottom) 3	oop for receiver

If the module has not already been loaded, the Squeak VM will attempt to "load" the named plugin, in this case ExtensionPluginName, in a manner that will vary depending upon the operating system. Squeak then attempts to find and execute the function **setInterpreter,** passing to that function a pointer to the Squeak VM interpreter proxy. (The standard plugin code saves this value in a global variable named **interpreterProxy.**) If this process succeeds, Squeak will then attempt to locate the pointer for the named primitive function, in this case, **primitiveName.** Should any part of this process fail, the VM will cease attempting to load the extension, and proceed by executing the Smalltalk code that followed the primitive specification.

If all goes well, however, control is passed to the named primitive function. If the primitive fails, that is, sets successFlag to false, the primitive must leave the stack intact (or restore it) before returning. If the primitive does not fail, then the primitive must pop the parameter and receiver oops from the Smalltalk stack, and must then push a valid oop thereon to serve as the return value.

This is among the most critical concerns when writing a plugin, and the most common cause of unpredictable behavior. The failure to comply with these linkage conventions can lead to substantial undefined behavior and will likely freeze or crash the Squeak VM.

When the primitive returns from execution, the interpreter will check successFlag. If the primitive failed, then control is passed to the corresponding Smalltalk code. Otherwise, the Smalltalk stack will be popped once (and only once) to obtain an oop, which will in turn serve as the answer for the primitive message send.

A primitive manipulates the Smalltalk stack (as distinct from the C function and parameter stack) through the interpreter proxy, using the functions described in the next section.

Primitive Access to the Interpreter Stack

The named primitive function may manipulate the stack using the following functions:

```
interpreterProxy stackValue: offset          interpreterProxy->stackValue(offset)
        interpreterProxy pop: nItems                 interpreterProxy->pop(nItems)
        interpreterProxy push: oop                   interpreterProxy->push(oop)

interpreterProxy                             interpreterProxy->popthenPush(nItems,oop)
        pop: nItems
        thenPush: oop
```

The method **stackValue:** offset answers the value on the Smalltalk stack offset slots from the top. Accordingly,

```
oop ← interpreterProxy stackValue: 0.
```

returns the value at the top of the stack. The method **pop:** removes the top nItems elements from the top of the Smalltalk stack, and answers the oop for the last value so removed. Conversely, **push:** pushes its parameter onto the Smalltalk stack. Method **pop:thenPush:** removes the nItems, and then pushes the specified oop onto the stack.

Since the named primitive rarely knows at the outset whether it will succeed or fail, it is uncommon for the primitive to pop values from the stack, leaving them in place for a "hasty retreat" by a simple return upon identifying a failure condition. The primitive is far more likely, therefore, to use **stackValue:** rather than the pop-related routines to access its parameters. The interpreter proxy provides various functions that facilitate this process by, not only loading the oop at the specified location, but in a single step also validating the shape of and converting the oop to C-friendly values, the parameters and the receiver. They are:

```
interpreterProxy stackIntegerValue: offset     interpreterProxy->stackIntegerValue(offset)
interpreterProxy stackObjectValue: offset      interpreterProxy->stackObjectValue(offset)
interpreterProxy stackFloatValue: offset       interpreterProxy->stackFloatValue(offset)
```

For variable objects or objects with instance variables important to the primitive, the oop would be loaded using **stackValue:,** and then validated or converted in turn. Finally, the proxy provides a mechanism to ease conversion of the C-friendly values to oops and pushing the corresponding oops onto the stack in a single step. They are:

interpreterProxy pushBool: cValue	interpreterProxy->pushBool(cValue)
interpreterProxy pushFloat: cDouble	interpreterProxy->pushFloat(cDouble)
interpreterProxy pushInteger: intValue	interpreterProxy->pushInteger(intValue)

Miscellaneous Plugin Tricks

The interpreter proxy provides a number of additional features useful for developing primitive plugins.

Success and failure of a plugin

The following routines are useful for establishing the failure of a primitive:

interpreterProxy failed	interpreterProxy->failed()
interpreterProxy primitiveFail	interpreterProxy->primitiveFail()
interpreterProxy success: aBoolean	interpreterProxy->success(aBoolean)

The first, **failed** returns true whenever the primitive has failed. **primitiveFail** establishes that the primitive has failed. **success:** establishes that the primitive has failed if the Boolean expression is false, and does not change the status otherwise.

Strict Type Checking

It is desirable from time to time to verify if a parameter or receiver is of a specific Class, rather than merely to confirm its Smalltalk shape. To determine if an oop represents an object that is an instance of a particular class, use

```
interpreterProxy                    interpreterProxy->isMemberOf(oop,aString)
        is: oop
        MemberOf: aString
```

To determine if an oop represents an object that is either an instance of a particular class or one of its subclasses, use:

```
interpreterProxy                    interpreterProxy->isKindOf(oop, aString)
        is: oop
        KindOf: aString
```

Determining the number of instance variables

Slang does not provide a method for determining directly the number of instance variables. It does provide a method for determining the total number of Smalltalk instance slots in an object:

interpreterProxy slotSizeOf: oop	interpreterProxy->slotSizeOf(oop)

For non-variable objects, **slotSizeOf:** will return the total number of named instance variables. For variable objects, however, **slotSizeOf:** returns the number of named instance variables *plus* the number of indexable variables. Accordingly, for variable objects, the number of named instance variables is given by:

```
(interpreterProxy slotSizeOf: oop) – (interpreterProxy sizeOf: oop)

interpreterProxy->slotSizeOf(oop) – interpreterProxy->sizeOf(oop)
```

Some care must be taken in using this expression, because **sizeOf** is not defined for most non-variable objects.

Instantiating Objects Inside a Plugin

Since **SmallInteger** objects require no memory other than the oop itself, they can be created, so to speak, on the fly. All other objects require the allocation of memory. While it is typically preferable to allocate objects used in a primitive in Smalltalk with a wrapper around the calling procedure, it is sometimes convenient or necessary to do so inside the primitive. The interpreter proxy provides routines for doing so. Given an oop representing a class, you can instantiate an object of that class and obtain an oop pointing to its object header as follows:

```
interpreterProxy                          interpreterProxy->
        instantiateClass: classPointer            instantiateClassindexableSize(
        indexableSize: size                               classPointer,size);
```

This operation does not execute initialization code typically associated with the object class, however. It merely allocates the space and initializes all instance variables to nil, akin to the Object>>**basicNew** method. To assure that an object is properly initialized, you might instead use:

```
interpreterProxy->clone(prototype)        interpreterProxy->clone(prototype);
```

which will perform a shallow copy of the object referred to by the prototype oop, as though the **clone** message were sent to the object.

It is fairly straightforward to obtain an oop for a class inside a plugin. At the outset, you can pass the class as a parameter to the function. Alternatively, given an oop, you can obtain the oop for its class using:

```
interpreterProxy fetchClassOf: oop        interpreterProxy->fetchClassOf(oop);
```

Alternatively, you can directly obtain the oop for certain fixed classes using any of the following messages:

```
        classArray                classLargePositiveInteger
        classBitmap               classPoint
        classByteArray            classSemaphore
        classCharacter            classSmallInteger
        classFloat                classString
```

Finally, the interpreterProxy provides a special-purpose function to facilitate the creation of objects of class **Point**.

```
interpreterProxy                          interpreterProxy->
```

makePointwithxValue: xValue yValue: yValue	makePointwithxValueyValue(xValue, yValue);

Memory Management and Garbage Collection

Garbage collection can be provoked from a primitive using the following functions:

interpreterProxy fullGC	interpreterProxy->fullGC();
interpreterProxy incrementalGC	interpreterProxy->incrementalGC();

which are analogous to the similarly named methods of class **SystemDictionary**. However, garbage collection can also occur whenever a new object is instantiated, either through the express instantiation above, or by using a method that can indirectly create a non-**SmallInteger** object, such as **positive32BitIntegerOf:**

When that occurs, any oop other than a **SmallInteger** oop stored in a C variable is invalidated and must be "reloaded," for example by obtaining new pointers from the stack or receiver. However, not all oops can be reloaded in this way, for example oops that were created by explicit instantiation.

To preserve oops across operations that can cause a garbage collection, the interpreterProxy provides a special *remappable oop* stack, which will hold references to oops which will be remapped during a garbage collection, so that the oop can later be reloaded. Functions are provided for pushing and popping values from the stack:

interpreterProxy popRemappableOop	interpreterProxy->popRemappableOop()
interpreterProxy pushRemappableOop: oop	interpreterProxy-> pushRemappableOop(oop);

For example, if temporaries or globals oop1 and oop2 held oop references to objects, the following Slang code would safely preserve the validity of the objects across a garbage collection.

```
interpreterProxy pushRemappableOop: oop1;
interpreterProxy pushRemappableOop: oop2;
. . . Smalltalk code that might result in a garbage collection . . .
oop2 ← interpreterProxy popRemappableOop;
oop1 ← interpreterProxy popRemappableOop;
. . . references to oop1 and oop2 . . .
```

Callbacks

One of the great weaknesses in the Squeak VM memory model is that the interpreter cannot readily be called from a C-language function. Accordingly, applications requiring callbacks are somewhat difficult to implement. A limited callback capacity can be approximated using the interpreterProxy method:

interpreterProxy signalSemaphoreWithIndex: semaIndex	interpreterProxy-> signalSemaphoreWithIndex(semaIndex);

Calling this method sends a signal to a Semaphore that had been registered using the Smalltalk>>registerExternalObject:.method.[3] The callback is set up by: (i) forking a process which waits on the Semaphore before calling the callback code, (ii) registering the Semaphore with the VM. Signaling the Semaphore from a C language routine using the interpreter Proxy can thereafter trigger the callback.

A Plugin for Swapping Blocks in a Word Processor

The Need for Speed

Programmers often need to move a blocks of data from one location in a data structure to another. For example, a user of a word processor may wish to move a block of text from one location to another, for example:

1. These
2. Lines
3. Out
4. Of
5. Should
6. Not
7. Be
8. Order

A user might wish to move lines 3 and 4 after line 7, to form the sentence "These Lines Should Not Be Out Of Order." This problem might be described another way, as the problem of "swapping" two unequally sized blocks of words, that is, swapping the two-word block {Out Of} with the three-word block {Should Not Be}. There are many ways to address this

[3] The #signalSemaphoreWithIndex: method does not immediately send the signal, but registers a request for the signal with the VM. The VM will send the signal shortly after the primitive returns control to the VM.

problem. For example, data structures such as linked lists can facilitate a speedy implementation of this operation.

If we are constrained, for whatever reason, to chose a more compact data structure, say a contiguous array of pointers to objects or an array of ASCII bytecodes, the problem becomes somewhat more interesting. A solution attributed to the authors of the TECO text editor is to reverse unequal portions of the string “These Lines Out Of Should Not Be Order” by performing the steps of:

1. reversing the elements of the first block in place;

   yielding: “These Lines fO tuO Should Not Be Order”

2. reversing the elements of the second block in place;

   yielding: “These Lines fO tuO eB toN dluohS Order”and

3. reversing the elements of the first and second blocks, taken together;

   yielding: “These Lines Should Not Be Out Of Order.”

This approach reduces the problem of swapping unequal blocks to the problem of reversing blocks in place. We might code this approach in Smalltalk by extending **OrderedCollection** with the following methods:

swapBlockFrom: firstFrom withBlockFrom: secondFrom to: last
“Modify me so that my elements from the block beginning with index, firstFrom, up to but not including the index, secondFrom –1, are swapped with the block beginning with index, secondFrom, up to and including the index, last. Answer self.”

```
self reverseInPlaceFrom: firstFrom to: secondFrom-1.
self reverseInPlaceFrom: secondFrom to: last.
self reverseInPlaceFrom: firstFrom to: last
```

reverseInPlaceFrom: from to: to
“Modify me so that my elements from the index, from, up to and including the index, to, are reversed. Assume that I am mutable. Answer self.”

```
| temp |
0 to: to - from // 2 do:
        [:index |
         temp ← self at: from + index.
         self at: from + index put: (self at: to-index).
         self at: to-index put: temp]
```

These methods work with all mutable subclasses of **OrderedCollection**, for example class Array. Executing the following DoIt:

```
#(this collection out of should not be order)
     swapBlockFrom: 3 withBlockFrom: 5 to: 8
```

which will answer:

(this collection should not be out of order)

These methods appear to work well and are general enough to handle all forms of **OrderedCollection**. However, the code can be noticeably slow when the blocks grow large. Consider a word processor maintaining documents as an Array of objects, each object representing one line. A line object might be an actual String of text in memory; an object representing a proxy to a file or intermediate file in which the text can be found; or some other object of relevance to the program.

Using that representation, one might implement both block moves and line insertions with **swapBlockFrom:withBlockFrom:to:.** However, for reasonably large files (about 100,000 lines or so), an insertion could take seconds to execute. In an interactive program, such a perceptible pause for a common operation could be intolerably slow. While many alternatives should be weighed to speedily perform this operation, a pluggable primitive is one that may save the day.

Step One: Designing the Interface

Before writing a primitive, it is useful to consider how it will be called. Doing so permits you to better understand the requirements and assumptions under which the primitive can be written. Since the document is represented as an Array of objects, we will consider overriding Array>>reverseInPlaceFrom:to:[4] with a primitive. We start by subclassing InterpreterPlugin to hold the plugin and primitive method:

```
InterpreterPlugin subclass: #FlipCollectionPlugin
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'VMConstruction-Plugins'
```

We then create a stub method in FlipCollectionPlugin, named primReverseFromto, and add a primitive interface to class Array:

```
primitiveReverseFrom: from to: to

(1)     <primitive: 'primReverseFromto' module: 'FlipCollectionPlugin'>
(2)     ^FlipCollectionPlugin doPrimitive: 'primReverseFromto'
```

Line (1) identifies the primitive to be called, and the module name expected. If the primitive executes properly, the return value (the value at the top of the stack upon completion) will be answered. If either: (a) the module cannot be loaded; (b) the primitive cannot be found in the module; or (c) the primitive failed when executed, then Line (2) will be executed. Line (2) sets up an environment to simulate in Smalltalk the primitive

[4] While it may be obvious in this case that the block move spends most of its time in the reverseInPlaceFrom:to: method, this is not always apparent in general. Squeak provides excellent tools for profiling performance, for example **MessageTalley**, and these measurements will often inform the question how a plugin should be designed.

execution, and executes it. The result of that simulation will be returned as the answer to the primitive.

We can then override reverseFrom:to: in class Array with a call to the primitive:

```
Array>>reverseFrom: from to: to

        ^self primitiveReverseFrom: from to: to
```

This completes the interface. We are now ready to build our primitive.

Step Two: Coding the Primitive

As discussed above, the primitive will be a parameterless method, which in this case we will call **primReverseFromto.**

```
primReverseFromto

        | from to rcvrOop rcvr t |

(0)     self export: true.
(1)     self var: #rcvr declareC: 'int *rcvr'.

(2)     to ← interpreterProxy stackIntegerValue: 0.
        from ← interpreterProxy stackIntegerValue: 1.

(3)     rcvrOop ← interpreterProxy stackObjectValue: 2.

(4)     rcvr ← self
                cCoerce: (interpreterProxy firstIndexableField: rcvrOop)
                to: 'int *'.

(5)     interpreterProxy success: (from >= 1 and: [from+1 <= to]).
        interpreterProxy success: (to <= (interpreterProxy stSizeOf: rcvrOop)).

(6)     interpreterProxy failed ifTrue: [^nil].

(7)     rcvr ← rcvr - 1.  "adjust for 1-based indexing."

(8)     0 to: to-from/2 do:
                [:index |
                 t ← rcvr at: from + index.
                 rcvr at: from + index put: (rcvr at: to-index).
                 rcvr at: to-index put: t].

(9)     interpreterProxy pop: 3 thenPush: rcvrOop
```

A brief discussion follows:

Line (0) assures that a C language function generated from this method will be an exported public reference. All primitives should have this declaration.

Line (1) assures that the C language variable associated with the Smalltalk temp *rcvr* will be declared as a pointer to integers. The default is type int.

Lines (2) load the parameters from the stack into temporaries. As you will recall, a method call first pushes the receiver onto the stack, followed by the parameters in ascending order. Thus, parameter **to** will be at the top (index 0) of the stack, followed by parameter **from** and the receiver.

Line (3) loads the oop for the receiver (an array of objects) into temp rcvrOop.

Line (4) loads rcvr with a pointer to a “void *” pointer to the first indexable address of rcvrOop, and coerces the result to “int *”.

Line (5) performs some bounds checking, resetting the success flag on a bounds failure.

Line (6) checks the success flag. Upon failure, the stack is left as-is, so that the primitive’s alternative code can be executed.

Line (7) adjusts rcvr for 1-based indexing. C-language arrays are 0-based, so by adjusting the pointer, subsequent references may treat the array as though it were a 1-based Smalltalk array.

Lines (8) perform the actual work. This is the same code set forth in the example code for **OrderedCollection**>>reverseFrom:to:, except that *self* is replaced with *rcvr*. (Without line 7, the code would have to be modified for C’s 0-indexed arrays.)

Line (9) pops the parameters and receiver oops from the stack, and pushes back an oop representing the receiver. (Since we wish to return the rcvr, a simple pop: 2 would have sufficed.)

Step 3: Building the Plugin

Having coded the primitive, we can directly test this code, albeit in slow-motion, using the plugin interpreter. Because the plugin is not installed, the primitive call in **primReverseInPlaceFromto** will fail, and call the interpreter. In this way, you can test most plugin code without actually compiling and installing the plugin.

Once we are satisfied that the code is working correctly, we can build the plugin. The following doIt will generate a C-language file corresponding to the plugin:

FlipCollectionPlugin translate

The generated file can then be compiled, using native system tools, as a shared library, and installed as a plugin for testing as live system code, resulting in a substantial and measurable speedup in the block move routines.

Sample Table of Contents

Sample Index