SISC: A Complete Scheme Interpreter in Java

Scott G. Miller <scgmille@cs.indiana.edu>

20th February 2003

Abstract

We will examine the design of SISC, a fully R5RS compliant heap-based interpreter of the functional language Scheme, with proper tail-recursion and first-class continuations. All noteworthy components of the interpreter will be described, and as well as discussion of some implementation details related to interpretation optimizations and efficient use of the Object-Oriented host language Java.

Contents

1 Introduction

SISC[1] is a Java based interpreter of the Scheme. It differs from most Java Scheme interpreters in that it attempts to sacrifice some simplicity in the evaluation model in order to provide a complete interpretation of Scheme. In particular, the complete R5RS Scheme standard is supported by SISC without sacrificing speed[2].

SISC uses a heap-based approach to control-context management, rather than a stack-based approach (using the procedure-call mechanism of the host language, and thus its native stack, to maintain continuations), and thus has full control over language continuations. For this reason, SISC is able to provide first-class continuations as defined in the standard, not just a subset of continuations such as the escaping continuations often provided by stack-based interpreters in Java. For a more thorough discussion of heap and stack-based interpreters, refer to [1].

SISC's engine is able to provide support for the core language features. With the addition of the portable syntax-case macro expander[3], SISC is able to provide hygienic macro support as per the standard.

Lexical environments are used in SISC to provide efficient searchless linear time lookups and sets (linear with respect to lexical depth) for lexically referenced variables. Top-level or global variable lookups and sets are constant-time through the use of inlined storage location pointers to store free variable references and syntactic definitions.

[1] SISC's predecessor was a stack-based interpreter named LISC (Lightweight Interpreter of Scheme Code), so, **S***econd* **I***nterprete*r *of* **S***cheme* **C**ode.

[2] There are a few exceptions to the standard in SISC that are not violations of the standard. In particular, SISC allows some syntax that the standard does not. For example, SISC does not require vector constants to be quoted, an extension common to many implementations.

SISC's speed is also quite competitive with respect to other Java Scheme interpreters. Since SISC compiles Scheme s-expressions to their fully-expanded forms, and stores them in an object-oriented internal representation, programs typically perform significantly faster than other interpreters that evaluate s-expressions in place. Later we will compare SISC to two other popular Java-based Scheme interpreters using publicly available benchmarks.

SISC also implements a full number stack, with support for fixed and arbitrary precision integers, arbitrary precision floating point numbers, rational numbers with arbitrary precision components, and complex numbers with arbitrary precision floating-point components. All the number forms are printed correctly, and all R5RS standard mathematical operations function correctly on these forms.

2 Rationale

When creating an interpreter for Scheme, the question of design goals is often raised. Many implementations of Scheme strive to prove truly useful for real-world use (Chez, MIT, Guile). Others are created for teaching purposes (DrScheme). Others seem designed as 'toy' implementations with interesting but irrelevant design goals (SILK[3], Scheme-in-one-Defun). Such endevours show the broad range of possibilities and the flexibility of Scheme as a small and elegant language.

Often, these implementations sacrifice some important features to meet their goals. But are the right sacrifices made? The two most frequently ignored requirements of the Scheme standard seem to be proper tail recursion and full first-class continuations.

2.1 Proper tail recursion

Proper tail recursion is not just an academic excercise. One of the central pillars of functional programming is the idea of recursive problem solving. It is the ability of functional languages to encourage this type of problem solving that is its greatest strength.

Consequently, programmers of functional languages use recursion to solve problems in ways that would ordinarily be tasked to iteration in other languages. A programmer assumes in such circumstances that their programs will run regardless of the size of their data, or the number of loops through a recursive function. Broken or nonexistant tail call optimization cripples a fundamental benefit of functional languages. SISC's realization of proper tail recursion is described in section 3.5.

2.2 Continuations

Continuations are less clear cut. Useful programs can be written without ever being aware of their existence. Nevertheless, their inclusion in a language as a controlable entity removes a significant barrier to producing programs that need more advanced control flow.

[3]SILK has since grown up, and renamed itself to JScheme. Its execution model remains the same, however.

First-class continuations allow for progamming constructs not possible in most langages today such as threading, co-routines, engines, and exceptions, without requiring explicit language support for those features. Along with Scheme's hygenic syntax extension, continuations make Scheme one of the most flexible languages in existence. SISC's realization of first-class continuations is described in section 3.6.

3 The Interpreter

3.1 Expression compilation

Before a Scheme expression can be evaluated, it must be compiled into SISC's internal expression representation. Do not put to much stock into the term 'compiling' as it is used here. S-expressions are merely parsed for semantics and turned into an internal representation that is used by the SISC evaluation engine. This process entails syntactic-expansion followed by the examination of the expression and the creation of the internal representation.

Syntactic expansion is accomplished using a portable macro-expander package by R. Kent Dybvig, et al [3]. This is a fully hygienic macro-expander that takes Scheme lists representing expressions and returns Scheme lists representing their fully expanded form. The expander is implements the algorithm described in [2]. After syntactic expansion, only the core Scheme expression types (BEGIN, LAMBDA, IF, QUOTE, DEFINE, variable references/sets, and applications), remain.

Then the expression is run through the compiler, which performs the task of examining the Scheme expression to create a number of SISC Expressions. The recognition of expression types such as LAMBDA, IF, DEFINE, etc. and the generation of internal representations to handle these forms is performed once at compile time. Consequently, we are able to realize a substantial performance increase at runtime, since we no longer have to examine the expressions to determine how to evaluate them. This combined with the object-oriented nature of SISC's Expressions means that apart from the dynamic method dispatch, no examination overhead (list-traversal, string comparison, etc) is necessary at runtime.

The downside is that a lengthy expansion and compilation must occur when s-expressions are entered or loaded. Due to the complexity of the macro expander, this does incur a significant overhead. For example, to load the macro-expander, the standard macro-definitions, the derived expressions and library functions, SISC must churn away for 10-15 seconds at startup[4]. This is an unreasonable amount of time to wait for a Scheme system to start, but is easily solved by loading a pre-expanded and compiled heap at startup.

The actual conversion from Scheme expressions to SISC Expressions is not worth describing in detail except for a few small cases. Quoted expressions merely compile to the value themselves (since Values evaluate to themselves). IF expressions with constant tests are eliminated, selecting the correct branch at compile time. Finally, some optimization is done to eliminate unnecessary begin expressions, and unnecessary expressions present in effect context.

[4]Depending on the platform, JVM, and hardware, of course.

3.2 Environments

SISC has two types of environments. The first, associative environments, create the foundation for top-level, free variable references. The second, lexical environments, support lexically scoped variable references within Scheme programs.

3.2.1 Associative Environments

Associative environments store a mapping between a symbol and a storage location. The storage locations are represented as an array of Values.

At compile time, each free variable reference is translated into a SISC Expression which is aware of which AssociativeEnvironment the variable is referencing. In addition, the expression contains an offset index into that AssociativeEnvironment's Value array. Initially, that offset is unset.

At runtime, when a free variable reference is encountered, the value is retrieved by retrieving the value present at the storage location associated with the reference. Notice that no lookup of any symbol is required, so the operation completes in constant time, with a very small overhead (one memory indirect). If the storage location was unset (as all references initially are), the offset is determined by querying the AssociativeEnvironment by the variable's name. If unbound at runtime, an error is signaled. If bound, the offset is set in the FreeReferenceExp, so further uses of the reference will not require a lookup in the environment.

Note also that lookups into the top-level environment are themselves constant time (with a fixed overhead), since the mappings are stored in a hashtable. The penalty is greater for such a lookup, however, because one must calculate a hash code, indirect into the table to fetch the storage location, then indirect to the storage location to retrieve the value. Fortunately, this happens at most once for each free-variable reference encountered at runtime.

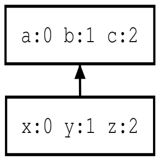
Setting of a free-variable is equally straightforward. Like FreeReferenceExps, FreeSetExps also contain the correct AssociativeEnvironment of the variable. A free-variable set then merely changes the value at a given offset in that environment. The same mechanism is used to determine the offset from an unresolved reference. If the storage location was not yet created, it is created at runtime and a new entry binding the variable to the new storage location is placed in the known environment.

3.2.2 Lexical Environments

All lexical references are resolved at compile time. This involves the compiler keeping an environment of all bound variables. As the compiler traverses the code, it builds the lexical reference environment, which consists of a number of ribs of formal parameters, each with pointers to a parent rib. For example, when processing the expression:

```
(lambda (a b c)
  (lambda (x y z)
     (+ a x c z)))
```

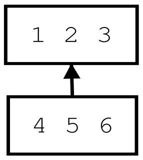
The following rib structure is present in the compiler:



When the compiler then encounters a reference to a variable, it looks up the variable in its lexical reference environment, keeping track of the number of ribs it crosses. For the variable *x*, it would cross no ribs, and find the variable *x* at position 0 in the rib. This produces the lexical reference 0:0. For the variable *c*, it would cross one rib, and find the variable *c* at position 2 in the rib. This produces the lexical reference 1:2. '1' is the lexical depth, while '0' is the lexical position.

If it does not find the variable in the lexical reference environment, it is a free variable reference, and generates a FreeReferenceExp whose semantics are described in section 3.2.1.

At runtime, the application of closures will modify the active lexical environment, creating a new lexical environment with the values of the formals in the LAMBDA, and a pointer to the parent lexical environment. To perform a lexical reference or a lexical set, one simply follows the parent pointer in the current lexical environment and decrease the depth by one until the depth of the reference is zero, then offsets into the lexical environment values at that point by the position field of the reference. For example, using the above Scheme expression, the following lexical environment might be active:



When the lexical reference *0:2* is encountered (representing the variable *z*), the parent pointers are followed until the depth is zero. The depth started at zero, so no pointers are followed. Then, we offset into that rib at position 2 to retrieve the value 4.

When the lexical reference *1:0* is encountered (representing *a*), the parent pointers are followed until the depth is zero. One parent pointer is followed. Then, we offset into that rib at position 0 to retrieve the value 1.

3.3 Evaluation

As a heap-based interpreter, the SISC virtual machine provides six registers, and allocates call-frames to store the state of the volatile registers when control-context is built. The registers are:

1. Next Expression (NXP): Holds the next expression to evaluate.

2. Accumulator (ACC): Holds the result of a previously evaluated expression.
3. Value-rib (VLR): Holds one or more evaluated arguments to a function application.
4. Lexical Environment (ENV): Holds the lexical environment active for the next-expression.
5. Failure Continuation (FK): Holds the active failure continuation, invoked if an error occurs.
6. Stack register (STK): The topmost call-frame.

The call-frames store volatile registers (those that may change due to the execution of an expression). To be precise,, the NXP, VLR, ENV, FK, and STK are the *volatile* registers which are stored in a call-frame.

A call-frame is essentially a data-structure representation of the stack. However, because we have explicit control over the stack of call-frames, we can treat each call-frame as the continuation at some point in the execution of an expression. If we take care to design our expressions such that they are immutable, then capturing a continuation is only a matter of returning the active call-frame (and packaging it such that it behaves like a procedure in Scheme).

3.3.1 Execution

The interpreter's main-loop will attempt to repeatedly evaluate the NXP register, as long as an expression is present in it. If the NXP register is unset, the main-loop will attempt to pop a call-frame from the stack (which may fill the NXP and cause the interpreter to continue).

Call-frames will be pushed onto the stack by the various expressions that are evaluated (which are described in section 3.3.2).

If at some point the NXP register and the STK register are both empty, the main-loop will terminate, returning the contents of the ACC register as the result of the evaluation of the entire expression.

3.3.2 Expressions

Expressions are defined in SISC as anything that can be evaluated, either for identity or to produce another expression. For all intents and purposes, every bit of data processed in SISC is an expression. Values, for example, are treated as expressions that evaluate to themselves.

The interesting expressions, of course, are those that implement the semantics of Scheme itself. In SISC, expressions lie in the package sisc.exprs, and fall into one of two categories:

*Expr	Expressions which set up the registers to process some next expression.
*Eval	Expressions which do 'post-processing' on the registers, to finish some *Expr type.

This distinction will become more clear as we examine each expression type in detail. We'll start with the simplest expressions; those that deal with variable references, the binding of variable locations, and the storing of values in storage locations.

First, lets examine EvalExp, the generic evaluation expression.

EvalExp

EvalExp is generated by the compiler to contain two Expressions, the *pre* and *post* expression. EvalExp will push the post expression onto the stack, and set the next expression register to *pre*. The consequence is that the result of the *pre*-expression is available in the accumulator when the post expression is evaluated.

Variable references and binding expressions

FreeReferenceExp

FreeReferenceExp will place the value of a top-level variable in the accumulator. If the variable is unbound, an error is raised.

LexicalReferenceExp

LexicalReferenceExp will look up a lexical variable (see section 3.2.2) and place the value in the accumulator. Unlike a free variable reference, it is impossible for a lexical reference to refer to an unbound variable, as all lexical references are discovered at compile-time. Variable references that aren't lexically scoped are automatically compiled as free variable references.

FreeSetEval

A FreeSetEval expects an evaluated value in the accumulator. When the FreeSetEval is evaluated, it will set a top-level variable to the value found in ACC.

LexicalSetEval

LexicalSetEval will set the value of a lexical variable to the value in the accumulator.

DefineEval

DefineEval handles top-level defines in SISC. DefineEval is constructed with a Symbol, and when evaluated, creates a top-level binding from the provided Symbol and the value present in the accumulator.

Control-flow expressions

Control-flow expressions include the expressions that handle BEGIN and IF. That is, they evaluate one or more expressions in a particular order, and return a value that is the result of the entire expression.

The Scheme begin expression is simply created by chaining a number of EvalExps together, each representing one expression in the Scheme begin. The post-expression of the last EvalExp (i.e. the last expression in the begin) becomes the value of the entire expression.

IfEval

IfEval expects a value in the accumulator, placed there by an EvalExp whose pre-expression was the test expression of the Scheme IF. If the value is non-false, it will place the true expression in the NXP. If the value is false, it will place the false expression in the accumulator. Either way, one of the branches becomes the result of the entire IF expression.

Function application

Function application is handled by the AppExp/AppEval pair in coordination with the FillRibExp.

AppExp

AppExp will push an AppEval onto the stack, then push a FillRibExp onto the stack with the operator and operand expressions, then set the NXP to the last operand. The FillRibExp will fill the VLR with the values of each operand, then set the NXP to the operator. Finally, control will return to the AppEval, which will conclude with the application of the operator to its operands.

AppEval

AppEval expects the evaluated operator to be present in the accumulator, and the values of each operand to be present in the value-rib. It will then apply the operator to the operands (what this entails is determined by the operator, see section 3.4).

Interestingly, AppEval is the only singleton expression. Since AppEval requires no context at all, the same AppEval can be used in all instances of an application.

Other expressions

A few more expressions are needed, to handle the creation of closures, and the evaluation of more than one expression.

LambdaExp

LambdaExp will create a closure by capturing the current lexical environment (in the ENV register), and placing the new closure into the accumulator. If the LambdaExp was compiled from a Scheme LAMBDA with dot notation (i.e. *"(x y . z)"*), the LambdaExp that is created will have a flag set called *infinite-arity.* This affects how the lexical environment is created when the closure is applied.

FillRibExp

FillRibExp expects to find the value of an expression in the accumulator. When constructed, it is given a pointer to a position in the value rib, a number of expressions, and a final or tail expression.

It sets this position in the value-rib to the value in the accumulator, and if more expressions remain to be evaluated, pushes another FillRibExp onto the stack with the next value-rib address (in an arbitrary order). When no more expressions remain to be evaluated, it places the tail expression into the NXP.

ApplyValuesContEval

The ApplyValuesContEval is produced by the evaluation of a call-with-values expression. It expects a single value or a multiple values object in the accumulator, and places the values in the value-rib. After which it places a procedure in the accumulator and sets the NXP to an AppEval. The result is that the value(s) that were in the accumulator are used as arguments to a procedure.

3.4 Procedures

Procedures differ slightly from expressions in SISC. They are applied to one or more values in the value-rib, and may cause additional expressions to be evaluated, by pushing additional call-frames onto the stack and/or setting the NXP to a new expression. There are three classes of procedures in SISC.

3.4.1 Built-in procedures

Built-in procedures are procedures whose behavior is determined by native code in a SISC module (see section 7). A built-in procedure will call the module with a procedure id; an integer which specifies which procedure is being applied. SISC's primitives are defined as built-in procedures with the host module sisc.Primitives.

3.4.2 Closures

Closures are user-defined procedures generated by LAMBDA expressions (specifically, the LambdaExp described in section 3.3.2). They consist of a closed lexical environment and a body expression. When applied, lexical bindings are made to the values present in the VLR. This is an efficient operation, since this only entails creating a new lexical environment with the current environment as the parent, and a copy of the VLR as the values.

If the closure is an infinite arity closure (see section 3.3.2), then the lexical environment will contain values for the required parameters, and a list containing the optional parameters as the last value position.

Finally, the ENV is set to this new lexical environment and the NXP is set to the body expression.

3.4.3 Continuations

Continuations (simply call-frames in SISC) are the final procedure class. When a continuation is captured, all the call-frames in the stack clone the value-ribs at each level[5], and mark each frame as captured (see section 6 for the reasoning).

When a call-frame is applied, all of the frame's registers replace the interpreter's registers, and the values in the VLR are placed in the accumulator. If more than one value was present, multiple values are placed in the ACC. (see section 4.3)

3.5 Proper tail recursion

Proper tail recursion comes easily as a side effect of the heap-based register architecture. All relevant Expressions are implemented with a knowledge of which subexpressions are in tail position and which are not.

Take for example the Expression tree created by an IF expression:

```
(if 1 2 3)
=>
(EvalExp 1
  (IfEval 2 3))
```

The EvalExp is aware that its *pre* expression is always in non-tail position, and that its *post* expression is in tail position. In this example, the 1 is in non-tail position, so EvalExp pushes the IfEval onto the stack and sets NXP to 1[6]. After 1 is evaluated, the stack is popped and the IfEval finds itself in NXP.

The IfEval is aware that both its *consequent* and *alternate* expression are in tail position. In this case no stack pushes are necessary. The appropriate expression (in this case the 2) are put into the NXP and control returns to the main-loop.

Begin expressions behave properly because they are built from numerous EvalExp's chained together. Naturally, the last EvalExp's *post* expression will behave as if in tail position, and thus the entire begin will have correct tail semantics.

Closure applications (the Closure expression) always bind the parameters in a new Lexical environment, then set the NXP to the body, again without pushing anything on the stack.

Applications are somewhat more complicated, since an application may or may not be in tail position. At compile time, an AppExp is determined to be either in tail or non-tail position (based on the expression that contains it). If an AppExp is in non-tail position, the current CallFrame (continuation) is saved to prevent corrupting the Value rib. If the AppExp is in tail position no save is done and the current registers are co-opted for use by the AppExp.

3.6 Continuations

Like proper tail recursion, first-class continuations are an elegant consequence of the heap-based register architecture. At any point in evaluation of any expression, the STK

[5]To prevent semantic errors due to side-effects in the operands of an application.

[6]In fact, it recognizes that 1 is an immediate and simply places it in the accumulator.

register represents the continuation of the active expression. To implement first-class continuations, SISC need only capture the STK register, and ensure that the contents of the stack cannot be modified.

When a CallFrame is captured, its VLR is cloned, any volatile expressions locked, and its STK parent captured as well. This prevents later evaluation modifying the present VLR's and altering the results of applying the continuation in the future. See section 3.4.3 for a brief discussion of how CallFrames are made to behave as procedures and the details of applying a continuation.

3.7 Error handling

SISC is a bit unique when it comes to how errors or exceptions are processed. The Scheme standard does not specify a method to establish error-handling at the program level. There are various implementations of error-handling in Scheme implementations. Chez Scheme, for example, allows the user to set an error-handler, which is a procedure that takes the same arguments as the standard ERROR function. The defined error-handler then overrides the default handler.

The problem with such a mechanism is that the user has to either override the active error handler without the ability to call the previously defined error handler, or explicitly code in the ability to call the previous handler by storing it before redefining the new procedure. This solution is inelegant, as it does not allow the user to write modular code that handles its own errors but can still transparently raise errors to earlier handlers without knowledge of their existence.

3.7.1 The User's view

SISC's error handling is loosely modeled on Java's. The user can define a local error handler, which will be called when an error is raised within the subexpression being evaluated. This is similar to Java's *try/catch* mechanism for exception handling. SISC's interface to error handling is defined as follows:

procedure: **call-with-failure-continuation** *thunk failure-handler*

procedure: **call/fc** *thunk failure-handler*

> *thunk* is a no argument procedure containing an expression to be evaluated, *failure-handler* is a three argument procedure.
>
> *thunk* will be evaluated in the current continuation. If an error is raised during its evaluation, *failure-handler* will be called with three parameters: The string message of the error, the continuation active at the time of the error, and the failure continuation active at the **call/fc** site.
>
> Applying the failure continuation with the error-message and error-continuation (the first two arguments) would produce the same results as not having installed the new failure-continuation.
>
> Examples:

```
 (call/fc (lambda ()
            (error "bad things happened"))
          (lambda (message error-continuation parent-fk)
            message))
=> "Error: bad things happened" ; a string

 (call/fc (lambda ()
            (error "bad things happened"))
          (lambda (message error-continuation parent-fk)
            (parent-fk message error-continuation)))
=> Error: bad things happened ; an error is signaled

 (call/fc (lambda ()
            (+ (car '()) 3))
          (lambda (message error-continuation parent-fk)
            (error-continuation 4)))
=> 7
```

3.7.2 The Implementation

SISC's error model is implemented using the failure continuation register (FK) of each call frame. When a new error handler is installed, the FK register is set to the expression that implements that handler.

Conceptually, this means there are two stacks active at any point in computation. The first is the ordinary stack of continuations. The second is a stack of failure continuations. When an error is raised, the details of the error are applied to the failure continuation. That failure continuation may rethrow the error, which causes *its* FK register to be applied to the thrown error's details.

If at some point the failure continuation is not set (is null in Java), a fatal Java exception terminates the interpreter. SISC's Read-Eval-Print-Loop is written in Scheme, and uses the above mechanisms to ensure that no Scheme error escapes the REPL to prevent this catastrophic failure.

In cases where a REPL is not required, it may actually be desirable for the Interpreter to throw an uncaught error, knowing that the calling Java code can handle the error.

4 Datatypes

The full range of Scheme values are supported, and are largely straightforward. Three types: symbols, numbers, and the implementation of multiple values are worth discussion.

4.1 Symbols

Symbols are stored as Java strings. In order to provide the pointer equivalence of symbols in Scheme, SISC keeps a static WeakHashMap in memory that maps the string values to the Symbol objects. When a new Symbol is requested, the symbol table is consulted, and the symbol is not regenerated if it exists in the table. Thus, EQUAL? equivalent symbols will also be pointer (EQ?) equivalent.

The use of WeakHashMap allows the Java garbage collector to reclaim unreachable symbols, despite their existence in the symbol table.

4.2 Numbers

The full number stack is supported using a class called Quantity. The Quantity class contains the values of each number's components. Quantity supports all of the math operations (including the trancendentals and proper inexactness contagion) for the following types of numbers:

FixedInt	Fixed precision integers
Integer	Arbitrary precision integers (bignums)
Decimal	Arbitrary precision floating point numbers
Rational	Ratios with arbitrary precision numerators/denominators
Complex	Complex numbers with arbitrary precision floating-point components

Because SISC does not have fixed precision floating point numbers[7], the precision suffixes 's', 'f', 'd', and 'l' make little sense. SISC treats each of those suffixes as 'e', and creates the arbitrary-precision float as one would expect.

Whenever a new number is generated, it goes through a simplification process. Complex numbers are simplified to decimal numbers when the imaginary component is zero. Ratios are reduced to their simplest form, and to Integers if the denominator is one. Integers are reduced to FixedInts if they will fit into a 32 bit Java integer.

Likewise, the mathematical operators will produce appropriate results. For example, SQRT of a negative number will produce a complex result[8], and overflows of FixedInt computations are promoted transparently to Integers. The result is a full number stack that works to keep numbers in their simplest form for efficient computation, without sacrificing precision.

4.3 Multiple values

Multiple values are treated as a separate type (sisc.data.Values). Unlike normal values, a multiple value object *cannot* be evaluated for identity. This is because the evaluation

[7]In fact there are several Quantity implementations, each with different precisions for floating point numbers. In each, only one precision is used, so again the flags are ignored.

[8]SISC's SQRT does *not* try to produce exact results for exact square numbers. There is no practical limitation against doing so, however.

of a multiple-values object implies that the Values object is being used in a single-value context. Thus, the evaluation of the Values object always produces an error. A special expression, the ApplyValuesContEval expression, is the only form capable of receiving a values object. (see section 3.3.2)

5 Java: Pros and Cons

Writing Scheme in Java gives us a number of benefits related to the ability to structure the interpreter using object-oriented methodologies. Also, the status of Java as a garbage-collected language means that we are spared the complexity of doing our own heap-management.

We also have some negatives introduced both by the use of Java and the selection of a heap-based interpreter. In particular, instantiation costs of often created structures (especially the call-frames) are significant. We address this problem in section 6. In addition there is the oft-sited criticism of writing an interpreter for a language on an interpreter (the Java Virtual Machine). However, with advances in Just-In-Time compilers in modern JVMs, this cost is small enough to justify Java's advantages.

5.1 Object-Oriented benefits

The use of an object-oriented language allows us to couple data with behavior. This turns out to be remarkably beneficial when designing an interpreter. For example, we expect value types to be printed in certain ways. Also, we know that the evaluation of an expression involves not only the semantics of the expression, but the values of certain fields in that expression.

In SISC, unlike many other Java-based interpreters of Scheme, we choose not to use Java's native types to represent Scheme values. Instead, we create explicit classes for each Scheme value. These classes are SISC Expressions that extend a Value class. The Value class mandates that its subclasses implement a DISPLAY() and optionally a WRITE() method (the former producing output compatible with DISPLAY in Scheme, the latter compatible with WRITE). The Expression class mandates that its subclasses implement an EVAL() method, which requires that the expression evaluate itself to produce additional expressions or values.

SISC Values all share the same EVAL() method, which merely sets the accumulator to to *this* (i.e., all Values evaluate to themselves). Each Value implements the DISPLAY() method differently, producing the appropriate human readable form of the value. Some Values implement the WRITE() method, when the WRITE output of a value differs from the DISPLAY output. If the WRITE() method is unimplemented, DISPLAY() is used.

Expressions, with all of their different semantics, share a common interface as well; the EVAL() method. Because each is aware of its own semantics (implemented in its EVAL() method), the interpreter itself needn't concern itself at all with recognizing expression types. It merely calls the EVAL() of any expression in the NXP register as long as an expression is present. We then rely on method dispatch in Java to take care of the rest. The benefit is simple, elegant code.[9]

Similar approaches are taken in other areas. For example, call-frames are implemented with the CallFrame class, which serves not only as a container for the saved registers, but is also a Procedure expression, and knows how to correctly apply itself when the continuation is applied.

6 Optimizations

We have so far described the design and implementation of SISC strictly as it relates to evaluating Scheme code. SISC must also present a respectable level of performance to be considered in favor of a stack-based implementation.

Most of speed from stack-based interpreters is realized by using the native stack to represent control context. Consequently, it is difficult or impossible to realize Scheme's first-class continuations in their entirety[10]. Heap-based interpreters, which manage their own control context, can realize the full semantics of Scheme's continuations, but often do so at a performance penalty. SISC uses two major optimizations to minimize this penalty and ensure excellent performance in the presence of first-class continuations.

6.1 Call-frame recycling

The primary source of overhead in SISC, and in many heap-based interpreters, is the creation of hundreds of thousands of call-frames. These call-frames live for a short duration and then are freed (or in the case of Java, eventually garbage collected). Because of the relatively expensive cost of call-frame instantiation (as compared to native stack operations), and the incredible strain placed on the garbage-collector to reclaim this memory, call-frame instantiation acts as a significant bottleneck for performance.

In SISC, we counter this problem by recycling call-frames. Whenever a call-frame is popped from the stack, we reclaim it (holding some small number of call-frames at any given time and letting the rest be garbage collected). When a call-frame is required, we check to see any old call-frames can be re-used. If so, we instead re-assign the fields of an old frame and return it. By keeping as few as 20 old frames around when possible, we are able to eliminate over 99% of all the call-frame instantiations, as well as place less stress on the garbage-collector to reclaim the old frames.[11] Both translate to a noticeable increase in speed.

[9]One possible disadvantage is that we do not know the underlying method dispatch mechanism in a given JVM. Thus if dispatch is slow, then the interpreter will suffer as well. Fortunately, it is desirable to all Java developers that dynamic method dispatch be fast, so performance in this area should receive considerable attention.

[10]It is in fact possible for stack-based interpreters to implement real continuations through the use of stack 'tricks'. However, this is not possible in Java since it is impossible to access or control the stack in any way except through the use of exceptions. Exceptions only allow program flow to procede *outward*, which is the reason stack-based Java interpreters of Scheme only claim to support escaping continuations.

[11]Compiling the initial environment provides a meaningful benchmark to highlight this optimization:

	Number of instantiations	Number of bytes allocated	% of total
No Call-Frame recycling	8,785,884 call-frames	351,435,360	63.69%
With Call-Frame recycling	76,228 call-frames	3,049,120	1.50%

There is a catch. Because Scheme allows continuations to be captured during evaluation, we cannot always overwrite old call-frames. Doing so may invalidate a previously captured call-frame.

To solve this problem, each call-frame has a boolean flag called the 'captured' flag. When a call-frame is captured using CALL/CC or other mechanisms, the captured flag is set on every call-frame in the stack. When the frame is then popped, the captured flag is examined, and if set, may not be placed in the reclamation pool. It is simply discarded. Thanks to Java's garbage-collector, the frame's memory may be reclaimed if it becomes unreachable. If, however, the frame is reachable, its contents will be left intact, and continuations capturing the frame will behave properly.

6.2 Expression recycling

Another small source of overhead is the cost of instantiating a number of FillRibExps. This expression type must be instantiated once for every argument to a procedure application.

We would like to recycle these as well, but in order for continuations to behave properly, every expression on the stack must be immutable.

As in the call-frames, we allow ourselves significant flexibility up until the moment the expression is captured in a continuation. In FillRibExp, we overwrite the fields of the expression and re-stack it, rather than instantiate a new expression to place on the stack. We can do this whenever we like unless the expression has been caught in a continuation.

Again, as in each call-frame, we tag expressions that we would like to alter with a captured flag when they have been captured in a continuation. As long as they have not been captured, they can mutate their own fields and re-stack themselves. If, however, the captured flag is present, a new expression is stacked instead.

For additional speed, we also maintain a pool of FillRibExps similarly to the CallFrame pool. When it is necessary to generate FillRibExps, the pool is checked first.

6.3 Immediate fast-tracking

With memory management bottlenecks out of the way, the primary source of overhead then becomes the evaluation of arguments for application of a function. Fortunately, in many cases the arguments to a procedure are so-called *Immediates*, an expression whose value can be determined without evaluating any other expression.

Immediates include all values and variable references. The former evaluate to themselves, and the latter require only a lookup. The AppExp therefor includes some logic to *fast-track* immediates.

When an AppExp is evaluated, it procedes through the list of operands in one direction (the direction, like the order of operations, is unspecified). As long as the operand is immediate, it fetches its value and places it immediately into the VLR. The same is done for the operator. Additionally, each FillRibExp performs the same check. When a non-immediate is found, it is placed into the NXP register and a FillRibExp is pushed to handle the result.

In the best case, an application with immediate operands and an operator need never use a single FillRibExp. The result of evaluating the AppExp is simply to set NXP to an AppEval, with the VLR already filled with values.

7 Extensibility

SISC allows native extension of its functionality through the addition of SISC Modules. A Module is simply a Java class with an initialization method that provides access to an interpreter context. A typical module will use this to bind a number of builtin-procedures within the top-level environment.

Due to the extensive object-oriented nature of SISC, a module may not only introduce new native procedures, but can also include class definitions for new types. Several SISC modules exist already, providing bridging to and from the Java language, networking capabilities, debugging, and more, all without touching the SISC engine at all.

8 Performance comparison

To compare the performance of SISC to two other well-known Java Scheme interpreters: SILK and Skij, the Gabriel LISP benchmarks[4], available for Scheme, were used. Only tests that completed without error on all three systems were included in these numbers. All times are in milliseconds:

Scheme System	**CPSTAK**	**CTAK**	**Dderiv**	**Deriv**	**Destructive**	**Div-iter**
SISC	516	1,063	1,907	1,860	2,291	937
Skij	5235	25,843	7,432	3,283	15,761	5,685
SILK	5,947	23,132	11,227	9,179	33,966	13,387

Scheme System	**Div-rec**	**Fread**	**Fprint**	**TAK**	**TAKR**	**Total**
SISC	1,021	127	28	44	534	10,328
Skij	4,345	565	22	2,410	2,870	73,451
SILK	13,061	38	2,485	5,919	8,955	127,296

9 Conclusion

Scheme is a rare beast in the world of programming languages. Most recent languages have implemented once or twice, usually by the inventors of the language. Scheme, by contrast, has been implemented dozens of times in a number of languages, each with their own strengths and weaknesses.

Scheme in Java has had a bit of a rocky start. Few compilers for Scheme exist at this time, none as of yet complete. Interpreters are less rare, but most have been designed with very specific goals in mind (SILK, for example, with the primary design goal being code size, Skij; for scripting Java programs). Suprisingly, none of these primary goals have yet been "*to run Scheme programs, well.*"

Building on modern interpretation techniques and adding object-oriented programming styles, SISC shows that creating a complete implementation of Scheme in Java *is* possible. We can see that one does not have to make limiting design decisions in order to produce a useful implementation of the language, or to sacrifice completeness for the sake of performance.

References

[1] Three Implementation Models for Scheme. R. Kent Dybvig. University of North Carolina Computer Science Technical Report 87-011 [Ph.D. Dissertation], April 1987.

[2] Syntactic abstraction in Scheme. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Lisp and Symbolic Computation 5, 4, pp. 83-110, December 1993.

[3] Portable syntax-case expander. R. Kent Dybvig et al. Available at *ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lang/syntax-case.tar.gz*

[R5RS] The Revised^5 Report on the Algorithmic Language Scheme. Richard Kelsey, William Clinger, and Johnathan Rees, editors.

[SISC] SISC is a freely available program, released simultaneously under the GNU General Public License and the Mozilla Public License. Recent versions of SISC can be found at http://sisc.sourceforge.net.

[4] Gabriel LISP benchmarks for Scheme. Available at

ftp://ftp.cs.indiana.edu/pub/scheme-repository/code/lang/gabriel-scm.tar.gz