Common Larceny

William D Clinger
Northeastern University
will@ccs.neu.edu

ABSTRACT

Common Larceny is an implementation of Scheme for Microsoft's Common Language Runtime (CLR), which is part of .NET. Common Larceny interoperates with other CLR languages using the JavaDot notation of JScheme, generating the JavaDot interfaces just in time via reflection, and caching them for performance. All other differences between Common Larceny, Petit Larceny, and Larceny derive from differences in the compiler's target language and architecture: IL/CLR, ANSI C, or machine code. The Larceny family therefore offers a case study in the suitability of these target architectures for Scheme.

1. INTRODUCTION

Common Larceny is a new implementation of Scheme for Microsoft's Common Language Runtime (CLR), which is the execution engine for .NET [11].

Common Larceny is the third in a family of implementations that use my Twobit compiler [5]. The original implementation, which was named Larceny because that's what "Lars's implementation" begins to sound like after you've said it enough times, was used for research that tested algorithms for compiler optimization, first class continuations, and generational garbage collection [5, 6, 12, 13, 14, 16]. Larceny generated native code for the SPARC, and did not run on other architectures.

After Larceny was released in 1999, Lars Hansen wrote a new back end that generated moderately portable C code instead of SPARC machine code. This implementation was named Petit Larceny, and will be released for the first time this summer.

Common Larceny is a port of Petit Larceny to Microsoft's CLR. It generates Microsoft's Common Intermediate Language (IL) instead of C, has a runtime system written in C#, and provides JavaDot notation for interoperation with other CLR languages [1].

This paper describes the design and implementation of Common Larceny, emphasizing issues that arose from mismatches between the semantics of Scheme and the semantics of the Common Language System. The paper concludes with some rough preliminary data on the relative performance of the Larceny and PLT Scheme (DrScheme) families, mainly to show how strongly a compiler's target language can affect performance.

2. DESIGN GOALS

2.1 Goal: Interoperability

Unlike Larceny, which was conceived as a research implementation, Common Larceny is intended for writing real programs that run under .NET. This goal implies interoperability with other CLR languages.

2.2 Goal: Compatibility

At the same time, we want Common Larceny to remain compatible with Petit Larceny at the source level. This goal implies full support for Scheme's first class procedures, first class continuations, the full numeric tower, proper tail recursion, and so on. This degree of compatibility made the port easier and less risky, and should simplify the job of maintaining Common Larceny over time.

We also want to improve compatibility between the Larceny family of implementations and the PLT Scheme family, which includes DrScheme and MzScheme [18]. The PLT Scheme family emphasizes ease of use, pedagogy, tools for static and dynamic debugging, programming environment, graphical user interface, and most of the other things that programmers have come to expect from their programming language. Perhaps the main shortcoming of PLT Scheme is its performance: the entire system is based on an interpreter. That interpreter is fairly fast as interpreters go, and there is a MzC compiler that delivers worthwhile improvements for some modules, but PLT Scheme generally cannot match the performance of compiler-based implementations such as Stalin, Bigloo, Gambit, Chez Scheme, or the original version of Larceny.

A reasonable degree of compatibility between the Larceny family and the PLT Scheme family would accomplish two things: (1) programs developed in PLT Scheme, but in need of more speed, could be ported to Petit Larceny; and (2) programs developed in PLT Scheme could take advantage of the facilities provided by .NET via Common Larceny.

class	`System.Object.class`
constructor	`(System.Collections.Hashtable.  100)`
dynamic method	`(.ContainsKey ht key)`
static method	`(System.String.op_equality "abc" "def")`

instance field (fetch)
instance field (assign)
	`(.name$ x)`
	`(set-.name$ x "Solomon")`

static field	`(Scheme.Rep.SFixnum.maxPreAlloc$)`

Table 1: JavaDot syntax

We had hoped to port DrScheme to Common Larceny, but that turned out to require translation of several hundred thousand lines of C and C++ code into Scheme. (Much of this code appears to have been written in C/C++ out of fear that it wouldn't have run fast enough had it been written in Scheme and interpreted. We can't just leave the C/C++ code alone because much of it is low-level code that talks directly to the garbage collector or to other components that look very different in Common Larceny.) Joe Marshall has automated most of the syntactic translation from C/C++ to Scheme, but it is still necessary to clean up the translated code by hand.

2.3 Goal: Performance

We didn't expect Common Larceny to run as fast as Petit Larceny, but we hoped it would be at least as fast as MzScheme, which is about twice as fast as DrScheme.

3. JAVADOT

Despite many minor differences, Microsoft's Common Language Runtime is fundamentally similar to the Java Virtual Machine, so interoperation with the CLR languages is fundamentally the same problem as interoperation with Java.

Instead of reinventing a wheel, we implemented the JavaDot notation designed by Ken Anderson and Tim Hickey for JScheme [1]. As illustrated in Table 1, JavaDot notation uses a set of lexical conventions to refer to the names of classes, constructors, static fields and methods, and dynamic instance fields and methods:

- Identifiers that end in `.class` name a class.
- Identifiers that end with a period name a constructor for the class whose name precedes the final period.
- Identifiers that begin with a period name dynamic methods.
- Identifiers that contain embedded periods name static methods.
- Identifiers that begin with a period and end with a dollar sign name a procedure that returns the value of a dynamic instance field.
- Identifiers that do not begin with a period but end with a dollar sign name a procedure that returns the value of a static field.

For example, a Scheme programmer would write

```
(let ((ht (System.Collections.Hashtable. 100)))
  ...
  (if (.ContainsKey ht key)
      ...)
  ...)
```

where a C# programmer would write

```
System.Collections.Hashtable ht
    = new System.Collections.Hashtable (100);
...
if (ht.ContainsKey (key)) {
    ...
}
...
```

Scheme has no static type system, so the semantics of calling a method from Scheme is slightly different from the semantics of calling it from C#. In Scheme, the runtime types of the arguments are used to resolve the method, whereas the static types of the argument expressions would be used in C#. This difference seldom matters, but a less convenient mechanism can be used when it does matter.

The JavaDot notation is implemented by a collaboration between the `read` procedure, which marks symbols that have special significance when JavaDot notation is enabled; the macro expander, which rewrites JavaDot notation into calls to the runtime support for JavaDot; and the runtime support for JavaDot, which uses the CLR's reflection features to locate the `System.Collections.Hashtable` class and the `ContainsKey` method of the example above, transforming the arguments from their Scheme representation to the representation expected by the constructor and method, and transforming the results back into the appropriate Scheme representation. JavaDot interfaces and marshalling code are generated just in time, but are cached for efficiency.

Here is an interactive example of JavaDot notation:

```
> System.Object.class
#<System.RuntimeType System.Object>

> (.GetType System.Object.class)
#<System.RuntimeType>

> (define x (Scheme.Rep.Factory.makeFixnum 17))

; Note: x is not a fixnum.
; x is a Scheme representation of the C# object
; that represents the Scheme fixnum 17.
; That C# object has an intValue() method
; and a value field.

> (.intValue x)

17

> (.value$ x)

17

> (set-.value$! x 24)

Error: CLR-INSTANCE-FIELD-SETTER not found: value
```

The error occurs because the `value` field (of the C# object that represents a Scheme fixnum) is `const`.

Procedures written in Scheme can be passed to C# and called from C#. This callback pattern is common in .NET libraries.

4. IMPLEMENTATION

The Twobit compiler translates Scheme expressions or files into the assembly language for a euphonious but hypothetical MacScheme machine. Apart from guidance provided by a few tables, this part of the compilation process is the same for Larceny, Petit Larceny, and Common Larceny.

In Common Larceny, the MacScheme machine assembly code is translated into Microsoft's Intermediate Language (IL). This translation involves some peephole optimization, but not as much as in Larceny or Petit Larceny. At load time, the IL is JIT-compiled to native code by the CLR.

Microsoft's IL is at about the same level as MacScheme machine language, but IL was designed to implement a radically different style of language. This fact is responsible for most of the differences, apart from JavaDot, between Petit Larceny and Common Larceny.

4.1 Compilation in Petit Larceny

Here is the definition of a Scheme procedure that allocates test inputs for a sorting benchmark:

```
(define (rgen n m)
  (let loop ((n n) (l '()))
    (if (zero? n)
        l
        (loop (- n 1) (cons (random m) l)))))
```

The Twobit compiler macro-expands and alpha-renames this definition, performs various optimizations such as closure analysis and incremental lambda-lifting (which will add `m` as a new argument to the `loop` procedure), and eventually converts the definition into the A-normal form shown in Figure 1. Twobit's code generator then translates that form into MacScheme machine assembly language, which looks like this:

```
L1001
    .proc
    reg/op1/branchf internal:branchf-zero?,2,1004
    reg/return  3
L1004
    save        3
    store       0,0
    store       1,3
    store       3,2
    reg/op2imm/setreg internal:-/imm,2,1,7
    store       7,1
    setrtn      1006
    global/invoke random,1
    .align      4
L1006
    .cont
    load        0,0
    setreg      7
    load        2,1
    load        6,2
    reg/op2/setreg internal:cons,7,6,3
    load        1,3
    pop         3
    branch      1001,3
```

In Larceny, this is assembled directly to native code. In Common Larceny, a slightly different version of it is translated into IL. In Petit Larceny, a slightly different version of it is translated into the C code shown in Figure 2. That C code is just a sequence of calls to C preprocessor macros that macro-expand the MacScheme machine instructions into C statements that perform the actions associated with those instructions.

4.2 Mismatches between Scheme and the CLR

Scheme is an unusual language in several ways [15]:

- Scheme is purely object-oriented in the rather basic sense that all values are objects.
- Scheme performs integer arithmetic, instead of arithmetic modulo 2^N.
- Scheme provides first class continuations, instead of `try/catch/finally`.
- Scheme is block-structured and higher-order.
- Scheme guarantees the asymptotic space complexity of tail recursion [7].

The CLR is an unusual target machine in several ways:

- Unions of value and reference types are not expressible.
- Exceptions must follow a simple `try/catch/finally` model.
- Managed code cannot inspect the control stack.
- There is no support for block structure.

These properties of Scheme and the CLR interact in several unfortunate ways.

```
(let* ((.T14 (lambda (.n|1 .m|1)
               (define .loop|9
                 (lambda (.m|3|28 .n|10 .l|10)
                   (let ((.T2 (zero? .n|10)))
                     (if .T2
                         .l|10
                         (let* ((.T5 (- .n|10 1))
                                (.T7 (random .m|3|28))
                                (.REG2 .T5)
                                (.REG3 (cons .T7 .l|10)))
                           (.loop|9 .m|3|28 .REG2 .REG3))))))
               (let* ((.REG1 .m|1) (.REG2 .n|1))
                 (.loop|9 .REG1 .REG2 '()))))
       (.T15 (set! rgen .T14)))
  'rgen)
```

Figure 1: A-normal form of the rgen procedure.

```
twobit_label( 1001, compiled_block_2_1001 );
twobit_reg( 2 );
twobit_op1_branchf_612( 2, compiled_temp_2_2, 1004, compiled_block_2_1004 ); /* internal:branchf-zero? */
twobit_reg( 3 );
twobit_return();
twobit_label( 1004, compiled_block_2_1004 );
twobit_save( 3 );
twobit_store( 0, 0 );
twobit_store( 1, 3 );
twobit_store( 3, 2 );
twobit_reg( 2 );
twobit_op2imm_131( fixnum(1), 3, compiled_temp_2_3 ); /* - */
twobit_setreg( 15 );
twobit_store( 15, 1 );
twobit_global( 1 ); /* random */
twobit_setrtn( 1006, compiled_block_2_1006 );
twobit_invoke( 1 );
twobit_label( 1006, compiled_block_2_1006 );
twobit_load( 0, 0 );
twobit_setreg( 15 );
twobit_load( 2, 1 );
twobit_load( 14, 2 );
twobit_reg( 15 );
twobit_op2_58( 14 ); /* cons */
twobit_setreg( 3 );
twobit_load( 1, 3 );
twobit_pop( 3 );
twobit_branch( 1001, compiled_block_2_1001 );
```

Figure 2: C code generated for the rgen procedure.

4.3 Representation of Values

Consider the addition of 1 to an integer variable `x`. On the SPARC, `(+ x 1)` compiles into something like

```
0       taddcc  %r1, 4, %result
4       bvc,a   #32
8       (DELAY SLOT)
12      sub     %result, 4, %result
16      or      %g0, %r1, %result
20      jmpl    %globals + 1096, %o7    ! +
24      or      %g0, 4, %argreg2
```

where the first two instructions finish the job most of the time, and the remaining instructions are executed only when `x` is large. The simplicity and efficiency of this code relies upon a representation of exact integers as the union of two types, fixnum and bignum, where fixnums are immediate values whose two low-order bits are zero, and bignums are tagged pointers whose two low-order bits are not both zero.

That union cannot be expressed by Microsoft's Common Type System. More generally, the representation of Scheme values as a union of primitive types (fixnum, boolean, character) and pointer or reference types (most other types) cannot be expressed in the CLR target architecture, so we are forced to use a less desirable representation.

In Common Larceny, all Scheme values are represented as instances of a `SchemeObject` class (not its real name). Some instances—booleans, the most common characters, and the smallest exact integers—are preallocated.

This representation—like all other representations that were available to us—is bad for interoperability. Every call between Scheme and some other language will require translation of data between Scheme's pure OO representation and the representation used in other CLR languages. This translation can be performed automatically, but it is going to be inefficient.

Furthermore the standard debuggers provided by Visual Studio.NET and other development environments are likely to display booleans and numbers as pointers.

4.4 Control Structure

Smalltalk-80 provides block contexts [10]. Common Lisp provides continuable exceptions [19]. Scheme provides first class continuations [15]. None of these advanced control structures fit into the simple `try/catch/finally` model provided by the CLR, and we can't implement these control structures in the usual ways because managed code cannot examine the control stack [8].

When we were designing Common Larceny, we thought it would be impossible to implement first class continuations while using the CLR control stack. Because compatibility with certain other implementations of Scheme was an explicit design goal, we implemented our own control stack.

That too is bad for interoperability. We can't use the CLR convention with our own control stack, so inter-language and intra-language calls must use two different calling conventions. This is not as bad as it sounds, because we were already forced to translate data on every inter-language call, but it does have some additional consequences for debugging. Standard debuggers will not be able to display the Scheme control stack, and will not work properly when tracing or stepping Scheme code.

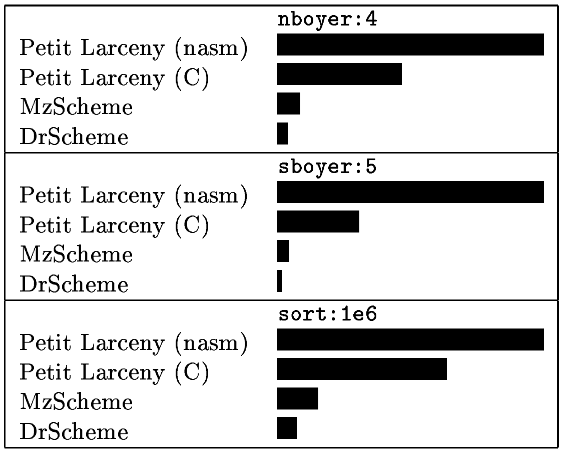

Table 2: Relative performance (longer bars are faster) for three benchmarks in Petit Larceny when compiled to native Pentium code (nasm) or C, and in interpreted MzScheme and DrScheme. Compiling to IL in Common Larceny delivers performance similar to that of MzScheme.

After most of Common Larceny had been implemented, three different researchers made independent discovery of a clever technique that would have allowed us to implement Scheme's first class continuations while using the standard CLR control stack with a slightly non-standard calling convention [17]. This technique works only in implementations that have certain special properties, but Common Larceny has those properties. The technique was discovered too late for Common Larceny, but may be of use to other implementors. In particular, the technique could be used to implement Common Lisp's continuable exceptions.

4.5 Block Structure and Tail Recursion

The CLR's lack of support for block structure did not affect Common Larceny because the Twobit compiler uses lambda lifting and closure conversion to convert most non-global variables into local variables, and allocates all remaining non-local variables on the heap anyway [5].

The CLR provides a `tail.` modifier to assist with the implementation of tail recursion, but we couldn't use it in the intended way because we aren't using the standard calling convention. At present, we use the `tail.` modifier for all intra-language calls, whether tail or non-tail, but a more conventional trampoline might be faster.

5. RELATIVE PERFORMANCE

From the limited benchmarking we have done so far, Common Larceny's JIT-compiled IL appears to run about as fast as interpreted MzScheme, which is usually about twice as fast as DrScheme.

We would like to use a single machine to compare the performance of Larceny, Petit Larceny, and Common Larceny, but Larceny runs only on a SPARC, and the current version of Common Larceny runs only in Microsoft's CLR under Windows. Lars Hansen has written a native code generator for the Intel Pentium, but it runs only under Linux at the moment.

In Figure 2 we compromise by showing the relative performance of Hansen's native code generator (nasm), Petit Larceny, MzScheme, and DrScheme. The performance of MzScheme can be assumed to approximate that of Common Larceny.

All three benchmarks shown in Figure 2 were run on a 2.8 GHz Linux machine with one gigabyte of memory. The `nboyer:4` benchmark is a modernized version of the old `boyer` benchmark from Gabriel's suite, with an exponential scale factor suggested by Bob Boyer [2, 3, 4, 9]. The `sboyer:5` benchmark is the next larger version of the same benchmark but with Baker's shared consing, which reduces gc time. The `sort:1e6` benchmark uses an efficient merge-sort routine to sort a list of one million fixnums.

6. STATUS AND FUTURE WORK

Common Larceny is currently in alpha release. We are fixing bugs, improving documentation, and simplifying the compile and build cycles so non-wizards can use them. The interpreter is very slow at present, but will eventually be replaced by incremental compilation to IL and on to native code.

Petit Larceny is being released for several non-Intel platforms now, with Intel versions to follow soon.

An approximation to MzScheme's module system is being implemented for the Larceny family. When that module system is operational, Larceny and Petit Larceny will provide a faster execution engine for compute-bound MzScheme programs, and Common Larceny will offer a migration path from MzScheme to .NET.

7. CONCLUSIONS

Scheme and Lisp can be compiled to run on the Common Language Runtime, and can interoperate with other CLR languages, but differences between Lisp/Scheme and the languages for which the CLR was designed constrain an implementation in ways that limit performance and reduce the usefulness of multi-language programming environments.

8. ACKNOWLEDGEMENTS

Lars Hansen wrote Larceny, Petit Larceny, and his native code generator for the Intel Pentium. Ryan Culpepper, Joe Marshall, Dale Vaillancourt, and others developed Common Larceny under the direction of Matthias Felleisen and myself. Felix Klock has been improving Larceny and Petit Larceny, and ran the benchmarks shown in this paper.

Development of Common Larceny was made possible by a generous gift from Microsoft. Experimental research using and extending Larceny has been supported by NSF grants CCR-9629801 and CCR-0208722, and by a Collaborative Research grant from Sun Microsystems.

9. REFERENCES

[1] Ken Anderson, Tim Hickey, and Peter Norvig. The JavaDot notation is described in `javadot.html` at `http://jscheme.sourceforge.net/jscheme/doc/`.

[2] Henry G. Baker. The Boyer benchmark at warp speed. *ACM Lisp Pointers* 5(3), July–September 1992, pages 13–14.

[3] Henry G. Baker. The Boyer benchmark meets linear logic. *ACM Lisp Pointers* 6(4), October–December 1993, pages 3–10.

[4] Henry G. Baker. Personal communication via electronic mail, 6 November 1995, quoting a personal communication via fax from Bob Boyer dated 3 December 1993.

[5] William D Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. Proceedings of the 1994 ACM Conference on Lisp and Functional Programming. ACM LISP Pointers VIII(3), July–September 1994, pages 128–139.

[6] William D Clinger and Lars T Hansen. Generational Garbage Collection and the Radioactive Decay Model. Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM SIGPLAN Notices 32(5), May 1997, pages 97–108.

[7] William D Clinger. Proper tail recursion and space efficiency. Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1998, pages 174–185.

[8] William D Clinger, Anne H Hartheimer, and Eric Ost. Implementation strategies for first-class continuations. In *Higher-Order and Symbolic Computation* 12(1), April 1999, pages 7–45.

[9] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* The MIT Press, 1985.

[10] Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

[11] John Gough. *Compiling for the .NET Common Language Runtime (CLR).* Prentice Hall, 2001.

[12] Lars Thomas Hansen. *The impact of programming style on the performance of Scheme programs.* M.S. Thesis, University of Oregon, 1992.

[13] Lars Thomas Hansen. *Older-first Garbage Collection in Practice.* Ph.D. Thesis, Northeastern University, November 2000. Available at `http://www.ccs.neu.edu/home/will/GC/lth-thesis/`.

[14] Lars T Hansen and William D Clinger. An experimental study of renewal-older-first garbage collection. In *International Conference on Functional Programming* (ICFP), 2002, pages 247–258.

[15] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 Report on the algorithmic language Scheme. ACM SIGPLAN Notices 33(9), September 1998, pages 26–76.

[16] The Larceny home page is at `http://www.larceny.org/`.

[17] Greg Pettyjohn, John Clements, Joe Marshall, Shriram Krishnamurthi, and Matthias Felleisen. Continuations from lightweight stack inspection. Submitted for publication.

[18] The PLT Scheme home page is at `http://www.plt-scheme.org/`.

[19] Guy L Steele. *Common Lisp the Language, 2nd Edition.* Digital Press, 1990.