Journal of Universal Computer Science, vol. 16, no. 2 (2010), 315-339
submitted: 19/10/09, accepted: 10/12/09, appeared: 28/1/10 © J.UCS

Using Lisp Implementation Internals

Unportable but Fun

Christophe Rhodes
(Department of Computing
Goldsmiths, University of London
New Cross, London, SE14 6NW, United Kingdom
c.rhodes@gold.ac.uk)

Abstract: We present a number of developer tools and language extensions that are available for use with Steel Bank Common Lisp, but which are perhaps not as well-known as they could be. Our motivation is twofold: firstly, to introduce to a developer audience facilities that can make their development or deployment of software more rapid or efficient. Secondly, in the context of the development of the Common Lisp language itself, we offer some observations of patterns of use of such extensions within the development community, and discuss the implications this has on future evolution of the language.
Key Words: Development tools – Language extensions – Language evolution – Lisp
Category: D.2.3, D.2.12, D.3.3

1 Introduction

Common Lisp the Language [Ste84] was published a quarter of a century ago; work done to improve and extend the language, in the context of the ANSI standardization process, finished fifteen years ago with the publication of ANSI document 226–1994 [PC94]. Since that time, the Common Lisp language as standardized through any formal process has not changed.

In that quarter of a century, the culture of programmers new to a given language appears to have changed somewhat: a number of single-implementation languages, where the current state of the implementation acts as the sole definition of the language's semantics (with the exception of discrepancies between the implementation and its documentation), have appeared and gained currency. Dynamic 'scripting' languages such as Perl, Python, Lua, and the like are by many metrics more popular than Common Lisp, and users of even single-implementation statically-typed languages such as GHC or Scala typically expect to find "batteries included" with their language, sometimes exhibiting difficulty with making the adjustment from a worldview where language and implementation are conflated to one where they are separate.

For example, a common complaint heard against the Common Lisp language is that it does not specify how to use OS facilities such as networking and threads. Yet it is unusual for programming languages with a formal standard to specify behaviour related to Operating System interfaces, networking facilities or other hardware devices[1]; the principal reason is that these interfaces are widely varied even at a single point in time, and essentially out of the control of the implementors of the language, and a common (hopefully static) subset of the interfaces is usually too poor for its standardization to be useful to users.

There is a tendency among Open Source Common Lisp developers to aspire towards portability as an end in itself. We will discuss some possible reasons for this tendency (which, we should make clear, is not an absolute) in section 4; for now, we wish to highlight some of the problems that this causes. One problem is that the desire for portability can act as a brake on development, because the ramifications of each development line on all platforms, including some which the developer may not even have access to, must be considered – not only slowing development down, but also tending to the use of only lowest-common-denominator functionality. In addition, time spent on portability is time not being spent on deployment, which is ultimately the driver of subsequent development. The counterbalance to this is that, particularly in the Open Source context, portability increases the reach of a given piece of software, and hence the potential number of contributors.

This feature of 'portability layers' allowing only lowest-common-denominator functionality can have unintended consequences. For example, the Universal Foreign Function Interface (UFFI) for Common Lisp, layered on top of implementations' high-level Foreign Function support, explicitly aimed to offer only the lowest-common-denominator featureset for interacting with 'Foreign' (non-Lisp) code. Though this software offered only limited functionality, it attracted the attention of developers because of its promise of portability across Lisp implementations, so while library bindings and applications using it were developed, the impression was given that Common Lisp in general could not give access to advanced foreign functionality. It took a significant effort, and the development of CFFI[2] to dislodge UFFI from developer consciousness.

It is not only users of the language who occasionally fall into this trap. For one reason or another, some portions of the Common Lisp standard are less polished and consistent than others. The chapter on pathnames [PC94, Chapter 19] attempted to specify a portability layer over the various different Operating System interfaces and functionalities; thus, there is support for a pathname *type* distinct from its name; multiple *versions* of a given pathname; and means to specify that a pathname can be found on a particular *host* or *device*. These concepts, modelled on the filesystem interface of the Lisp Machine, unfortu-

[1] But not unheard of: the 2005 revisions of Ada95 include access to many Operating System facilities, and even earlier versions of the Ada standard require implementations to provide access to hardware interrupts in a standard way.

[2] The Common Foreign Function Interface is based on simple, lowest-common-denominator *primitives* (such as access to a particular address in memory), and provides a UFFI-compatibility mode as well as greater functionality. `http://common-lisp.net/project/cffi/`.

nately do not map terribly well to the filesystems in use today; partly because of the attempt at portability across unknown systems, the test suite developed by Paul F. Dietz [Die05] to test specified behaviour covers relatively little, as almost no pathname operation is specified sufficiently unambiguously or consistently.

This paper discusses some little-documented (or completely undocumented) features of Steel Bank Common Lisp (SBCL), as a nudge away from the desire for portability as an end in itself and towards using the full range of an implementation's features. This of course does not remove the need for careful consideration of any deployment issues[3], nor is it intended as an argument for fragmenting an already small development community, but it is motivated by a desire to see innovation in tools and language extension be taken up and used, so that experience with them can be used to guide their further development. As a secondary objective, this paper can address some of the paucity of documentation. In the Open Source development world, the tension between documenting a facility to inform potential users and documenting a facility as an expression of support for that interface tends to lead to a general lack of documentation; the examples presented herein might pique the reader's interest enough to investigate further despite that lack.

We work with SBCL in this paper because that is the Common Lisp implementation with which we are most familiar. We hope, however, that the technical ideas presented here can transfer to other implementations; there are certainly similar tools and language extensions in other Lisp implementations, some overlapping in functionality to those presented here and some different in approach or effect. The social observations presented here are applicable to developers who use different Lisp implementations, though (obviously) not to all such developers; they have been made in an unsystematic way over a period of years, interacting with commercial and open source developers in person and online.

The rest of this paper is organized as follows: sections 2 and 3 are adapted from a tutorial presented at the 2009 European Lisp Symposium, held in Milan, and collectively provide an overview of some SBCL-specific functionality. Section 2 discusses some of the lesser-known tools within SBCL that can be used to assist in portions of the development cycle – tools whose use is primarily in the management of large systems, profiling and testing. Section 3 includes some miniature case studies of the use within application code of unportable language extensions, attempting to motivate their use by suggesting how they can save time and effort in the long run. Section 4 mentions other similar functionality in SBCL, and attempts to draw conclusions regarding the use of language extensions and the evolution of the language itself.

[3] As in the early history of the `reddit.com` startup, with initial implementation in Common Lisp, developed with one implementation on one Operating System and deployed with another implementation on a different Operating System.

2 Developer Tools

In this section, we discuss tools aimed squarely to assist the developer in the process of development. The first case relates to allowing users of subsystems with their own packages to have confidence that they are not opening themselves to an interaction problem in the presence of another third-party library. The second briefly covers profiling in its various different forms, and the third concludes this section with a discussion of a code coverage tool, which can be used not only to assess the coverage of a test suite, but also to assist in working out which pieces of a codebase are live code and which are never called.

2.1 Package Discipline

Programming in the large, as described for example in [Sei05, Chapter 21], involves dealing with the facts that it is perfectly reasonable for different software systems to want to give the same name to different concepts, and that these systems might need to interoperate. The way to address this in Common Lisp is through the use of packages, which control the lookup of symbols from their (string) names at read-time.

The Common Lisp standard specifies a number of actions on packages and symbols that the programmer must avoid, or else invoke undefined behaviour [PC94, Section 11.1.2.1.2]. These restrictions boil down to: not altering the structure of the `common-lisp` package (by removal of standard symbols or altering the home package of those symbols) and not pervasively altering the bindings or other meanings of standard symbols. The expressed rationale in the writeup of the `LISP-SYMBOL-REDEFINITION` X3J13 issue[4] was to clarify that an implementation is allowed to prohibit behaviour that would be tantamount to redefinition of system functionality.

The case of programming in the large, however, remains even with these restrictions on altering the `common-lisp` package. A simple scenario where this matters is as follows: a programmer wishes to build an application and depends on a third-party library; and the programmer's application package `:use`s that library package (in the `defpackage` form). Then, without reviewing the entirety of the library package's exports, *any* definition in the application package runs the risk of colliding with a definition in the library package. In some cases, this would be spotted quickly; `structure-class` redefinitions typically give immediate feedback, as do function definitions. However, redefinition of a `standard-class` is explicitly permitted by the standard, and there are other forms of rebinding or redefinition that need not give immediate feedback to the programmer, which might lead to confusing bugs later.

[4] A document not formally part of the standard, but reflecting the decision-making process of the standardization committee.

```
(defpackage "FOO"
  (:use "CL" "SB-EXT")
  (:export "FROB" "FROB-POP"
           "WITH-FROB-POP")
  (:lock t))

(in-package "FOO")
(defun frob () ...)
```

(a)

```
(defpackage "FOO"
  (:export "FROB" "FROB-POP"
           "WITH-FROB-POP")
  (:lock t)
  (:implement))
(defpackage "FOO-INT"
  (:use "CL" "SB-EXT" "FOO")
  (:implement "FOO" "FOO-INT"))

(in-package "FOO-INT")
(defun frob () ...)
```

(b)

Figure 1: Two styles of package definitions with SBCL-style package locks. In (a), a single package is used both as the working package for the definitions themselves and for the external interface (here, the symbols `foo:frob`, `foo:frob-pop` and `foo:with-frob-pop`. In (b), the `foo` package is the external interface only; the definitions are implemented within the `foo-int` package.

An additional wrinkle is that even if the application writer's definitions collides with no existing definitions in the library, there is still a potential problem: if the application introduces a new definition to an external symbol of the library, then this has the potential to collide with another, similar new definition in another body of code which uses the same library – and here there is the potential for truly hard-to-diagnose bugs. The way to prevent these is to strongly discourage modification – in any way – of symbols or packages that are not one's own.

One method of achieving this is the style of Common Lisp programming that does not `:use` or `:import-from` any packages beyond the `common-lisp` package, but always explicitly qualifies symbols. This style has its advantages, not least the fact that the identity of every symbol used is explicit, but can get unwieldy; a variant style uses `:import-from` only, allowing the programmer to audit definitions for a fixed set of names (rather than a large and potentially volatile set of names, the entire list of exported symbols from any used package).

In order to better support more usual styles of programming with packages (ones not requiring explicit listing or qualification of all symbols), SBCL provides

```
(defmacro with-frob-pop ((&key) &body body)
  `(macrolet ((frob-pop () ...))
     ,@body))

(defmacro with-frob-pop ((&key) &body body)
  `(locally (declare (disable-package-locks frob-pop))
     (macrolet ((frob-pop () ...))
       (locally (declare (enable-package-locks frob-pop))
         ,@body))))
```

Figure 2: In the presence of locked packages, macros which themselves provide local macros or functions overriding global definitions (such as the `with-frob-pop` macro, top) must locally disable the package locks around the rebinding, and then enable them around the user code (bottom).

a package lock extension, which allows the programmer to request that actions modifying a package or bindings of a package's symbols trigger an error, so that any potentially harmful name collision is caught early. The SBCL package lock facility is designed to be unobtrusive: where there would be no way for *users* of a locked package to write code without knowing about the presence of a package lock, errors in package discipline are not caught; for example, there is no way to specify that part of the interface to a package is binding particular restarts or using a particular symbol as a catch tag, and so at present these bindings are not treated as violations of the lock. Other than that caveat, the facility offers protections modelled after the restrictions in the Common Lisp standard on the modifications of standard packages [PC94, Section 11.1.2.1.2], but for arbitrary packages.

Figure 1 illustrates the setup necessary for declaratively locking a package (it is also possible to programmatically lock and unlock it after creation). The simpler case is where a single package is used, merely requiring the `:lock` argument to be used with `defpackage`. In the more complex case, an interface package `foo` is defined, but its functionality is constructed working within the `foo-int` package (which uses `foo`). In this case, `foo-int` is declared to *implement* the `foo` package (as well as itself). One detail is worth mentioning; sometimes a macro's functionality is provided by rebinding function or macro definitions locally, as schematically shown in figure 2. In this case, the implementor of the macro (not the user) must insert some declarations to inform the compiler about a lexical region of code where the rebinding of a symbol should not be prevented by a package lock.

2.2 Profiling in all its forms

It is often the case that the first draft of a program is not as fast as necessary or desired, and consequently needs optimizing for execution speed. It is now generally well-understood that a programmer's intuitions about the areas of the code which need optimizing are unreliable; a programmer's attention is generally drawn to the most pessimal routines (a static property of the code), whereas time spent executing the program is mostly spent in slow routines that are called many times (a dynamic property of the use of the software). Thus, the recommendation is always to profile a representative execution of the program before spending valuable programmer time in optimizing it, as time spent optimizing infrequently-called functions has negligible effect on the total running time.

In addition, many programmers do not have a reliable intuition for the performance characteristics of even relatively simple Common Lisp operators; because of different implementation strategies and different priorities in the facilities they provide, implementations themselves vary – sometimes quite significantly – in their performance. The benchmarks in [Gab85], along with community-maintained extensions, provide a broad view of an implementation's performance across a suite of tasks: this may give an indication of performance on a larger-scale programme, but rarely more than an indication.

A secondary use of measuring performance characteristics is in the investigation of unknown systems: both benchmarks [Rho04] and profiling data, particularly call graphs [GKM82] can provide a useful overview of the implementation of a system, a feature that profilers share with code coverage tools, discussed in the next section.

SBCL provides not one but two profilers: an instrumenting profiler [SE94], where function entry and exit are wrapped with code to keep track of time and other resources consumed by the wrapped function; and a sampling profiler [GKM82], which arranges for normal execution to be interrupted at a regular interval, recording at each interrupt the current program state.

The sampling profiler can be used to measure either CPU time or wall-clock time, or indeed to measure the allocation of pages; see figure 3 for an example function to profile and expected results. The instrumenting profiler adds overhead to each profiled function, introducing a systematic error in timing measurements, but can be more selective (profiling only certain selected functions or sets of functions) and repeatable (less vulnerable to sampling error). Both flat table and call graph reports are available, the latter only from the sampling profiler, which also integrates with the disassembler to show precise sample counts at each machine instruction.

We can contrast this with a portable profiler. While it is a testament to the stability and flexibility of the language that a portable profiler can be and has been built, and that a body of code unmaintained since 1994 runs with

```
(defun sb-sprof-example-fun (x y)
  (declare (type (unsigned-byte 16) y))
  (dotimes (i y)
    (setf x (mod (+ x x) most-positive-fixnum)))
  (sleep 0.01)
  (values x (mod x y)))

(defun sb-sprof-example (&optional (mode :cpu))
  (declare (type (member :time :cpu :alloc) mode))
  (sb-sprof:with-profiling
      (:mode mode :report :flat :loop t :max-samples 1000)
    (dotimes (i 200)
      (sb-sprof-example-fun 3 #xffff))))
```

Figure 3: Example code to demonstrate use of the statistical profiler in SBCL. `sb-sprof-example` can be called with an argument of `:cpu`, `:time` or `:alloc`, which should respectively identify `truncate`, `sleep` and `sb-vm::generic-+` (the low-level routine implementing two-argument addition of numbers) as the major users of the respective resources.

minimal modifications on Lisp systems released in 2009, it is also the case that the metering[5] portable profiler, by its portable nature, cannot provide all the functionality described above: a sampling profiler requires integrating with the system, and while an instrumenting profiler can be written portably it may interact badly with other system functionality, such as function tracing. This issue, and others, is discussed in more detail in [Lev09].

2.3 Code coverage

The language as standardized has endured (at the time of writing) for 15 years, and in that time whole paradigms in software engineering such as the Unified Process, Extreme Programming and Test-Driven Development have sprung up, gained currency, and in some cases largely disappeared again. Of course, since the Common Lisp standardization process to a large extent codified existing practice in the Common Lisp community since the 1984 language document [Ste84], and since that document itself was an attempt to produce a dialect of Lisp that was culturally compatible with many of the Lisp dialects active at the time, it should come as no surprise that there are substantial bodies of code portions of which are well over twenty years old that can run in contemporary Common

[5] One of the utilities by Mark Kantrowitz, available from the CMU AI repository.

```
(require :sb-cover)
(declaim (optimize sb-cover:store-coverage-data))
(asdf:oos 'asdf:load-op :cl-ppcre-test)
(cl-ppcre-test:run-all-tests)
(sb-cover:report "/tmp/cl-ppcre/")
```

Figure 4: A sequence of forms which if entered at the SBCL REPL will compile and load the cl-ppcre system [Wei09] with coverage annotations, execute the test suite, and generate reports of the code coverage, viewable with a web browser at `file:///tmp/cl-ppcre/cover-index.html` (see figure 5).

Lisp implementations; indeed, SBCL itself is one such, as the implementation can trace its history through its genesis in a fork from CMUCL [Mac92] in 1999 to CMUCL's origins in Spice Lisp from the early 1980s.

The long history of some bodies of code give the opportunity of the accumulation of quite some layers of cruft, which can cause significant difficulties in working out exactly what codepaths are relevant to a particular situation; maintainers (which includes the original developers of the software, given a suitable amount of time elapsed to allow for the fallibility of human memory to do its work) are often confronted with a large body of code, not much being relevant to a particular bug report or development avenue.

Code coverage tools can assist in this circumstance. The essence of a code coverage tool is to record and report which code has been executed during the course of some (possibly extended) interaction with the software. In SBCL's case, this is implemented by instrumenting each form within the code with instructions to set a flag within a cons, and a data structure mapping those conses to the source location of the associated code. It is then possible to report the forms which have or have not been executed during the interaction; the `sb-cover` module distributed with SBCL generates an html file for each file with instrumented code, displaying the coverage information using colour, with overview files to provide quick summaries and entry points into the reports.

Perhaps the canonical use of a code coverage tool, in contrast to the software archaeology use case presented above, is in the development or modification of a test suite. It is in this context that a previous code coverage tool for Common Lisp was developed and presented [Wat91], which also claims portability. However, that portability is achieved by shadowing defining symbols (`defun`, `defmacro`) and walking the bodies[6]; this portability comes at the expense of

[6] Rather than a full codewalk, the cover system (available from the CMU AI repository and described in [Wat91]) substitutes for atoms in bodies using `sublis`.

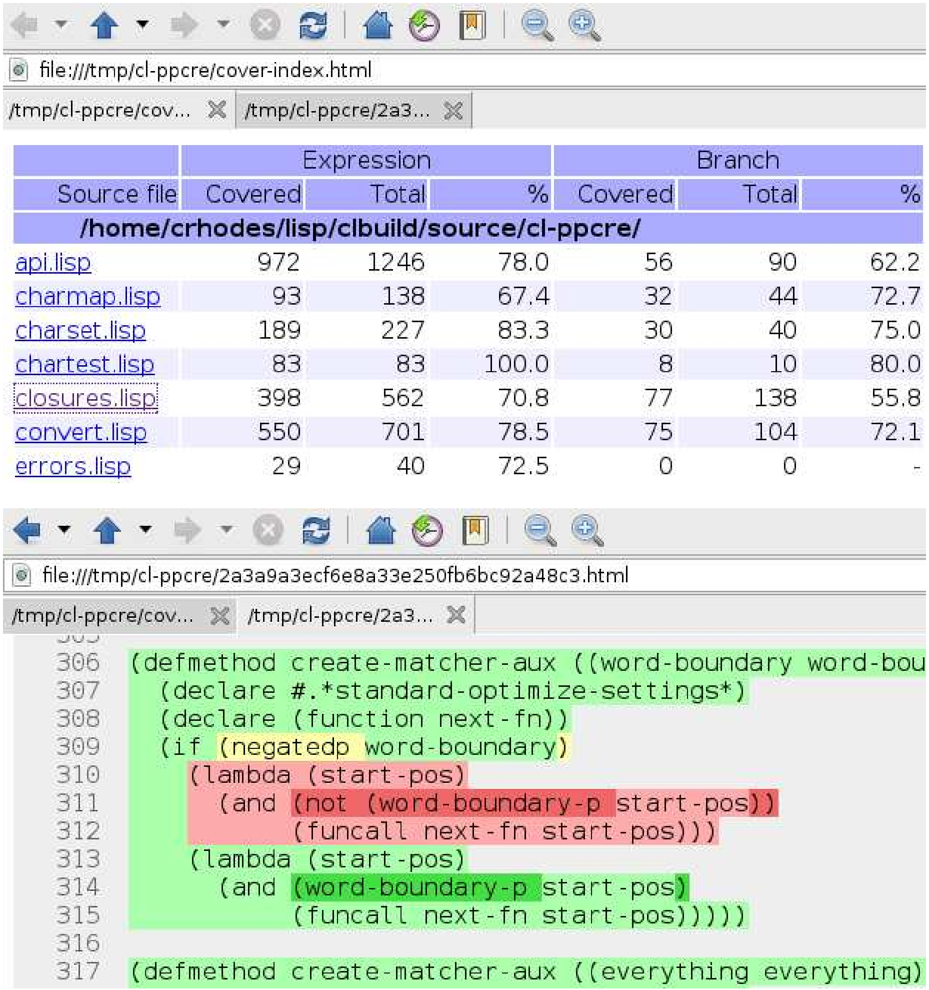

Figure 5: Illustrative output of `sb-cover`'s html output after running the test suite from cl-ppcre. The top panel shows the overview of the report, with summary data for all instrumented files; the bottom shows a section of the report for one particular file, with background colours indicating whether code was executed or not (light green / light red) and whether both, one or neither branch of a conditional was taken (strong green / yellow / strong red).

composability: it is not straightforwardly possible to compose multiple extensions of defining forms, so code using another such extension of Common Lisp using this shadowing technique (`series`, for example) could not use `cover` to assess coverage. By contrast, the `sb-cover` module in SBCL is integrated into the compiler, using a compiler optimization directive (see figure 4) and therefore composes transparently with other SBCL compiler hooks (single-stepping and other debugger facilities) and with user extensions.

3 Case Studies

In this section, we include some studies of functionality implemented in SBCL using particular unportable techniques. The first case study, implementing a symmetric encryption and decryption algorithm, relies heavily for its effectiveness on the compilation of a high-level description of the algorithm (in a language where the integer type is unbounded in size) to machine instructions. The second combines an efficient implementation of string equality with the ability to subclass the `specializer` metaobject class, to produce a generic function which dispatches efficiently between methods based on the `pathname-type` of its argument. The third case study implements a space-efficient representation of DNA base sequences as a subclass of `sequence`, allowing the user to use Common Lisp sequence functions on instances of the user-defined class.

3.1 Encryption

This case study demonstrates how to relieve the conflict between dynamic, run-time typing of objects – which necessarily means reserving some bits in a machine word to indicate type information – and many cryptographic algorithms designed to run efficiently on modern hardware, typically relying directly on the arithmetic operations provided by the CPU. The type tagging of objects in a dynamically-typed language means that the range of integers on which fast, hardware arithmetic can be performed is normally reduced; however, that range can locally be expanded to the full hardware width by explicitly requesting it – which can be done in portable code for unsigned arithmetic and logical operations.

The specific case covered in this case study is a complete implementation (source code for which is in appendix A) of RC5 encryption and decryption [Riv94]. In addition to efficient modular arithmetic, which we present in section 3.1.1, we also demonstrate how to extend the compiler itself by providing new low-level implementations of user-defined functions in section 3.1.2. While we present these techniques in the context of a cryptographic algorithm, they are also applicable to the implementation of hardware emulators (such as Nevermore[7], an emulator for the Explorer E1 Lisp Machine) and virtual machines (for example the Cloak[8] implementation of the Java Virtual Machine).

3.1.1 Modular Arithmetic

Many cryptographic algorithms, including symmetric and public-key encryption and cryptographic hash functions, are designed for relatively efficient implementation on general-purpose hardware. This means that the computational primitives making up the kernels of these algorithms are typically expressed in the systems of arithmetic provided by general-purpose arithmetic and logic units, which generally implement arithmetic with an implicit integer width equivalent to a masking operation after each operation – or, equivalently, arithmetic modulo 2^{32} for 32-bit ALUs.

There is an impedance mismatch here with many Lisp systems, in that the width of the ALU's integer arithmetic is also typically the pointer size, and dynamically-typed Lisps require some of those bits to act as tags to inform the Lisp runtime how to interpret the remaining bits; for example, SBCL uses 3 tag bits on 32-bit architectures, in such a way that the integers representable as immediates – `fixnums` – have the range $[-2^{29}, 2^{29}-1]$. Any integers outside this range in general must be represented as a pointer to a heap-allocated structure – a `bignum` – containing the bits representing that number.

However, within a single function, SBCL can temporarily represent integers using the whole machine width: either $[-2^{31}, 2^{31}-1]$ or $[0, 2^{32}-1]$ depending on the interpretation of the bits as representing a signed or unsigned integer. This fact, in combination with some observations about the distributivity of masking operations over many other arithmetic operations, allows for SBCL to compile the portable way of expressing arithmetic modulo 2^{32} – with an explicit mask or `logand` with `#xffffffff` – to the obvious machine instructions. This functionality, described in more detail in [DR04], appears not to be very common, perhaps because the combination of compilers emphasizing the speed of generated code and dynamically-typed languages is relatively rare; the closest analogue of which we are aware concerns the implementation of arithmetic operations on fields *smaller* than a machine word in Java [RR04].

Figure 6 demonstrates the effect of the support for hardware modular arithmetic in SBCL. The function `foo` computes a mathematical function of its three 32-bit integer arguments, and the result of the function is reduced modulo 2^{32} by the call to `logand`. Because all the functions used are arithmetic and logical primitives (and never shift bits downwards), the SBCL compiler can infer that

[7] `http://www.unlambda.com/nevermore/`
[8] `http://lichteblau.blogspot.com/2007/08/cloak.html`

```
(defun foo (x y z)
  (declare (type (unsigned-byte 32)
                 x y z))
  (logand (logxor (ash x 14)
                  (1+ (logandc2 y z)))
          #xffffffff))
```

```
  C0:  C1E70E   SHL EDI, 14    ; (ash . 14)
  C3:  8BCE     MOV ECX, ESI
  C5:  F7D1     NOT ECX        ; (lognot .)
  C7:  8BD3     MOV EDX, EBX
  C9:  21CA     AND EDX, ECX   ; (logand .)
  CB:  42       INC EDX        ; (1+ .)
  CC:  31D7     XOR EDI, EDX   ; (logxor .)
```

Figure 6: A somewhat contrived example function `foo` (top) and the x86 disassembly produced (bottom), with the type checks in the prologue and the tagging of the return value in the epilogue elided for clarity.

intermediate results also need only to be kept to 32 bits, and consequently all the computations can be done using hardware instructions. Note that there is no need to request high `speed` or low `safety` from the compiler for the arithmetic to be optimized: given the type declarations for the arguments `x`, `y` and `z`, the resulting machine code is both fast and safe.

3.1.2 Bitwise Rotation

The design of cryptographic algorithms also often attempts to 'mix' bits together. An operation commonly used to implement part of this mixing is bitwise rotation, which is not naturally expressible in terms of normal arithmetic primitives; whereas Common Lisp defines its mathematical operations over the integers, bitwise rotation only makes sense given a fixed-size bitfield.

Specifically, we can define 32-bit rotation operators for 32-bit integer i and 5-bit count k as

$$\mathrm{rotl}_{32}(i,k) = i \times 2^k + \left\lfloor \frac{i}{2^{32-k}} \right\rfloor \bmod 2^{32}$$
$$\mathrm{rotr}_{32}(i,k) = i \times 2^{32-k} + \left\lfloor \frac{i}{2^{k}} \right\rfloor \bmod 2^{32}$$

which are sufficient for 32-bit applications. Although SBCL provides a general facility for bitwise rotation in its `sb-rotate-byte` module, we use this opportu-

```
(in-package "SB-VM")

(macrolet
    ((def (fun internal opcode)
       `(progn
          (defknown ,internal ((unsigned-byte 32) (unsigned-byte 5))
              (unsigned-byte 32)
              (foldable flushable movable))
          (define-vop (,internal fast-ash-left/unsigned=>unsigned)
            (:translate ,internal)
            (:note "inline 32-bit rotation")
            (:generator 5
              (move result number)
              (move ecx amount)
              (inst ,opcode result :cl))) ; opcode: ROL or ROR
          (declaim (inline ,fun))
          (defun ,fun (x y)
            (declare (type (unsigned-byte 32) x y))
            (,internal x (logand y #x1f))))))
  (def rotl32 rotate-left/unsigned=>unsigned rol)
  (def rotr32 rotate-right/unsigned=>unsigned ror))
```

Figure 7: An implementation of 32-bit bitwise rotation operators for SBCL's x86 architecture backend.

nity to demonstrate in figure 7 how to implement specialized rotation operators for the x86 architecture corresponding to the definitions above. In that figure, for each of the left- and right-rotation cases, the macroexpansion makes the low-level implementation function known to the compiler, defines a virtual operation (a VOP, an object responsible for emitting machine code) implementing the rotation – the machine code to be emitted follows the `:generator`; VOPs in general also have type restrictions, documentation and other information, here mostly inherited from a VOP implementing left-shift – and provides a higher-level entry function which checks types and reduces the count modulo 32 before calling the low-level function.

3.2 Dynamically-compiled pattern matching

Pattern-matching is touted as a productivity gain, particularly by advocates of languages in the OCaml / ML / Haskell vein; there is significant literature on optimizing the pattern-matching dispatch [LFM01] and in providing language extensions to express pattern matching on more than declared datatypes [Xi03]. What with Common Lisp being a programmable programming language, it should come as no surprise that there exist pattern-matching libraries for use directly in Common Lisp. What might be slightly more of a surprise is that these libraries are generally of suboptimal quality, in the sense that the runtime of the

```
(defmacro string-case (string-form &body clauses)
  (let ((string (gensym "STRING")))
    `(let ((,string ,string-form))
       (cond
         ,@(loop for clause in clauses
                 if (typep (car clause) 'string)
                   collect `((string= ,string ,(car clause))
                             ,@(cdr clause)))))))
```

Figure 8: A naïve implementation of `string-case`, expanding into a `cond`, with each `string-case` clause resulting in a call to `string=`. For an efficient implementation of `string-case`, including minimization of branch instructions, see [Khu07].

code generated to perform the pattern matching is greater than necessary: reducing to a linear search, attempting to match the input to each pattern in turn (e.g. `match-case` in cl-unification; `match` in bpm[9]). This is particularly surprising given that implementations of CLOS naturally perform what is effectively optimized pattern discrimination when computing the applicable method of a generic function given a set of arguments.

This case study constructs a simple dynamically-compiled implementation of dispatch updatable at runtime over strings, compiling the dispatch on demand using a combination of an efficient `string-case` macro [Khu07] and extensible method specializers [NR08]. The motivating example is in the implementation of a generic viewer (modelled by a `view` generic function) which accepts a pathname and dispatches to an appropriate viewer based on the `pathname-type` of that pathname.

3.2.1 string-case

While dispatch to the appropriate effective method of a generic function given arguments is one application of pattern matching, it is not the only one; pattern-matching in other languages is also often used to dispatch on different tuple or list structure or on the contents of such structured data. One example that is occasionally desired in Common Lisp is a `case` form with a non-`eql` predicate; for example, a `string-case` macro where the keys are tested against the datum with `string=` (see for example an elegant solution by Erik Naggum for this case [Nag01]).

[9] `http://github.com/nallen05/bpm`

Figure 8 shows a simple implementation of the `string-case` macro as an illustration of the intended semantics (though in this simple implementation we do not support multiple keys per clause for reasons of space). This implementation, while typical of the code in Common Lisp pattern-matching libraries, has poor performance characteristics in that the time for performing the dispatch to the appropriate code is $O(N)$ in the number of keys. Since the keys in this case are assumed known at compile-time, a pattern-matching compiler can optimize this dispatch into something more like a tree search, as in [Khu07].

3.2.2 Extensible specializers

The Common Lisp Object System (CLOS) provides programmers with the ability to construct functionality from individual pieces by defining appropriate methods of generic functions, which are later combined into an effective method for a particular set of arguments; in addition, this allows for the simple definition of protocols, whereby the user of a piece of software may trace, extend or customize its behaviour by calling introspective generic functions and defining extending or overriding methods on generic functions that are documented as providing the functionality of the software; indeed, CLOS itself can be implemented to support such protocols [KdRB91].

The standard set of specializers in Common Lisp is limited to two classes: the `class` specializer itself, which indicates applicability if the argument in that position is a generalized instance of that class (*i.e.* an instance of that class or its subclasses); and an `eql` specializer, indicating applicability if the argument is the same as the given object.

A recent extension to the set of metaobject protocols in SBCL's implementation of CLOS allows the programmer to define their own specializer classes for their own generic function metaobject classes [NR08]. This facility, which interoperates with the other CLOS machinery, can be used as in appendix B (where we implement a `pathname-type` specializer, applicable when the argument is a `pathname` whose `type` field is equal to the specializer's argument, to enable code of the form of figure 9 to be written. As with ordinary generic functions and methods, addition and removal of methods can happen at any time, and cause the discriminating function to be reset to its initial state, which compiles and installs an optimized dispatch function for itself when it is next called.

This facility allows users to define protocols interacting with objects other than through their classes or their identities, which in principle allows more flexibility in the organization of code, allowing the structure of the project rather than the restrictions of the object system to dictate how to express and implement the desired functionality. However, it is not yet convenient to write functionality using subclasses of specializers with SBCL; with class-based specializers, the MOP function `compute-applicable-methods-using-classes`

```
(defgeneric view (pathname)
  (:generic-function-class pathname-type-generic-function))

(defmethod view ((pathname (pathname-type "jpg")))
  (view-image pathname))
(defmethod view ((pathname (pathname-type "jpeg")))
  (view-image pathname))

(defmethod view ((pathname (pathname-type "png")))
  (view-image pathname))

(defmethod view ((pathname (pathname-type "htm")))
  (browse-url pathname))
(defmethod view ((pathname (pathname-type "html")))
  (browse-url pathname))
```

```
A19: 634101       MOVSXD EAX, [RCX+1]  ; (char . 0)
A1C: 4883F868     CMP RAX, 104         ; (char= . #\h)
A20: 0F8582000000 JNE L7
A26: 634105       MOVSXD EAX, [RCX+5]  ; (char . 1)
A29: 4883F074     XOR RAX, 116         ; (logxor . (char-code #\t))
A2D: 488BD0       MOV RDX, RAX
A30: 634109       MOVSXD EAX, [RCX+9]  ; (char . 2)
A33: 4883F06D     XOR RAX, 109         ; (logxor . (char-code #\m))
A37: 4809C2       OR RDX, RAX          ; logior
A3A: 63410D       MOVSXD EAX, [RCX+13] ; (char . 3)
A3D: 4883F06C     XOR RAX, 108         ; (logxor . (char-code #\l))
A41: 4809C2       OR RDX, RAX          ; logior
A44: 4885D2       TEST RDX, RDX        ; (= . 0)
A47: 7408         JEQ L4
```

Figure 9: Method definitions and a fragment of the `x86-64` disassembly of the dispatch portion of the generic function (specifically, the test for `"html"`, jumping to `L4` on success; the jump to `L7` proceeds to test for `"jpeg"`, a length-based test having already been applied). The discriminating function is invalidated on addition or removal of a method, and recompiled as needed.

is sufficient to enable cacheing of most dispatch functionality, but this protocol must be bypassed when using extended specializers. An analogous protocol function `compute-applicable-methods-using-specializers`, along with some supporting functions, might well provide this convenience, but has yet to be properly integrated in SBCL's implementation of CLOS.

3.3 Generic sequences

The Common Lisp language as standardized reflects its long history; there are several examples of behaviour that is specified generically, without providing any standardized means for the programmer to interoperate with this behaviour: thus, by the standard, `make-hash-table` accepts any of four standardized equality predicates, with no means to specify hashing functions and predicates; arithmetic functions work with reals and complexes, but there is no way to extend the `number` type (though it is not specified that `real` and `complex` form an exhaustive partition of `number`); similarly, there are many functions acting on sequences in general [PC94, Chapter 17], but no way for the programmer to add sequence types beyond the standard `vector` and `list`.

In [Rho07], the author describes a protocol for allowing the user to perform sequence operations on arbitrary user-defined CLOS objects, with the price being a minor deviation [Rho06] from the standard regarding error-signalling behaviour of `make-sequence` and related functions. This protocol provides the potential for convenient provision of sequence functionality on sequence-like data structures (such as gap buffers [SVM04] for interactive editors, or space-efficient representations of DNA sequences: see figure 10) without having to reimplement a large standard library. All sequence functions in SBCL have default implementations calling five basic generic functions in the `sequence` package; once methods on those generic functions have been implemented, any function from the sequences chapter of the ANSI standard works directly on instances of this class, providing for easy instance creation (for example by using `(coerce "GATTACA" 'dna-sequence)`, querying (through `count` or `search` for example) and serialization to file.

The implementation strategy in figure 10, representing the identity of the base by an integer between 0 and 3 but translating that into one of the characters `"ACGT"`, suggests the need for `sequence-element-type`, a protocol function additional to the ones suggested in [Rho07], for expressing the set of elements which can be stored in the sequence. For backwards compatibility, a default implementation returning `t` on user subclasses of `sequence` can be provided.

The presence of this extension makes it easy and straightforward to allow the user to call sequence functions that would otherwise typically be partially reimplemented; indeed, even for data structures that are not strictly speaking sequences (such as leaves of a binary tree) it can be convenient to make a portion of this functionality available – while for those which are sequences supporting some extra information, making the entire Common Lisp sequence functionality available lowers barriers to the implementation of functionality that would otherwise be tedious to provide.

```
(define-symbol-macro bases "ACGT")
(defun convert (x) (position x bases))
(defclass dna-sequence (sequence standard-object)
  ((data :type (simple-array (unsigned-byte 2) (*)) :initarg data)))
(defmethod print-object ((o dna-sequence) s)
  (print-unreadable-object (o s :type t)
    (prin1 (coerce o 'string) s)))

(defmethod sequence:length ((o dna-sequence))
  (length (slot-value o 'data)))
(defmethod sequence:elt ((o dna-sequence) index)
  (aref bases (elt (slot-value o 'data) index)))
(defmethod (setf sequence:elt) (new-value (o dna-sequence) index)
  (setf (elt (slot-value o 'data) index) (convert new-value)))
(defmethod sequence:make-sequence-like
    ((o dna-sequence) length &key initial-element initial-contents)
  (let* ((data (make-array length  :element-type '(mod 4)))
         (result (make-instance 'dna-sequence 'data data)))
    (cond
      (initial-element (fill result initial-element))
      (initial-contents (map-into result 'identity initial-contents)))
    result))
(defmethod sequence:adjust-sequence
    ((o dna-sequence) length &key initial-element initial-contents)
  (let ((data (slot-value o 'data)))
    (setf (slot-value o 'data)
          (apply #'sb-sequence:adjust-sequence data length
                 (cond
                   (initial-element
                    (list :initial-element (convert initial-element)))
                   (initial-contents
                    (list :initial-contents
                          (mapcar #'convert initial-contents)))))))
  o)
```

Figure 10: Implementation of a space-efficient sequence type to represent DNA sequences, taking two bits per character while preserving all standard Common Lisp functionality. The first four definitions implement the representation itself, while the method definitions on functions in the `sequence` package provide the sequence-like behaviour for instances of the class.

4 Conclusions

In the two previous sections, we have covered a range of developer tools and some language extensions available in SBCL; our survey here is of necessity incomplete – for example, we have not discussed effective use of the debugger and stepper; facilities for using atomic operations to write lockless threadsafe data structures; hooking into the type derivation phase of the compiler; extending the `sb-alien` foreign-function interface – but samples the spectrum of implementation-specific functionality available. We should note that all of the facilities presented here are compatible with the language standard, in the sense that they do not conflict with any mandated behaviour, with the exception of the generalized sequences extension described in section 3.3 which requires a minor deviation [Rho06].

Informal and highly unscientific straw polls suggest that while the developer tools discussed in section 2 are commonly used and appreciated by members of the SBCL community, even those primarily or exclusively writing for development and deployment with SBCL tend not to make extensive use of the language extensions such as those discussed in section 3, while the most common reason for not using a particular development tool is not knowing about its existence[10]. In addition, there is little enthusiasm for implementation extensions that break conformance with the ANSI CL standard; while users accept that there will be bugs in implementations, there is usually little enthusiasm for any proposal to alter an implementation's semantics to be deliberately incompatible with the standard (as opposed to merely providing functionality for behaviour left undefined).

These observations are understandable: in using an implementation's development tool, programmers do not necessarily make it more difficult for themselves to port their software to a different implementation later, as the code developed does not retain a dependency on the tools used to interact with it (with the minor exception, among the tools presented here, of package locks). Using an unportable language extension in developing, however, almost by definition makes it harder to port the software; the consideration has to be whether the potential benefits from being able to express a particular solution more easily outweigh the potential costs – not an easy consideration to make. The same goes, but even stronger, for using an implementation which deviates significantly from the standard; whatever the value of using a standardized language, an implementation's deviation from the standard must diminish it.

Anecdotally, the Open Source Lisp development community places a high value on code portability; one possible explanation for this is because the members of the Open Source community compete for eyeballs (in the sense of the phrase "given enough eyeballs all bugs are shallow") and for approbation, both of which would tend to increase with increased portability. While acknowledging the overlap between Open Source developers and commercial entities, we would suggest that this motivation is less strong in application-focussed development; however, it is not clear that there are enough commercial users of SBCL who make their development information public to draw conclusions about the use or otherwise of language extensions[11].

[10] Sometimes a certain lack of polish can be an impediment; an earlier draft of this paper contained a section on SBCL's stepper facility, but it did not work in a sufficiently reliable manner to be described.

[11] Again anecdotally, users of two commercial implementations of Common Lisp, Al-

Addressing this desire for portability is not straightforward; if an extension's functionality could be provided in a portable way, it is not truly a language extension. The obvious approach would be to provide a portable layer for as much of the extension's functionality as possible, along with a native implementation for some subset of Lisp implementations; examples of unportable libraries for which this has been tried include the Environments Access[12] extensions from Franz Inc.; this does not seem to have been used to any extent in code intended to run in implementations of Common Lisp other than Allegro, nor does there seem to have been a demand from users that this extension should be supported elsewhere. This might be because the portable layer provided insufficient value, and so programmers and implementors do not see the potential return on the investment of producing and using the full capabilities of the extension – or it might simply be that the adoption timescale in the Lisp world is naturally slow.

In contrast, one Open Source Lisp project which uses unportable implementation features and yet is widely used and actively developed is SLIME, the 'Superior Lisp Interaction Mode for Emacs': it started as a CMUCL-only project, with portability explicitly not being a goal; porting efforts were if not actively discouraged, then not encouraged either; however, SLIME now not only supports all reasonable Common Lisp implementations to a greater or lesser extent, but also Clojure, some Scheme implementations, and languages even further from its origins. The particular success of SLIME is at least partly attributable to the poverty of Open Source Lisp interaction environments at the time, however; the alternatives (ILISP, `readline`-based interfaces) were neither featureful nor robust. Nevertheless, it might well be that the approach of developing a strong set of features first and then dealing with porting made it clear that a credible alternative was being developed, and so helped attract developer attention.

The above has implications on future evolution of the Common Lisp language. While the prospects of a new formal standardization process in a standards body such as ANSI are dim, given the absence of deep-pocketed users and the relative lack of pressing need, the fact that there is some community desire for language evolution is undeniable: various efforts such as the Common Lisp Request For Implementation founder essentially on lack of administrative manpower. However, the best in language evolution can only come from an informed position, which means that extensions have to be tried and tested before being stabilized and proposed as an evolutionary step to the language. This need for experimentation, for testing is the primary motivation for our desire for users to use implementations' facilities as far as is practical, rather than staying within the standard comfort zone.

legro CL and Lispworks, seem highly attached to the respective unportable systems delivered with those implementations, such as AllegroGraph and CAPI respectively.

[12] `http://www.lispwire.com/entry-proganal-envaccess-des`

Acknowledgments

This paper uses and extends material presented at the 2009 European Lisp Symposium. The author wishes to make clear that SBCL is a collaborative development project which has received contributions of code from many tens of people (and bug reports, development ideas and other contributions from many more). The author thanks in particular Alastair Bridgewater, Paul Khuong, Tobias Rittweiler and Nikodemus Siivola for helpful suggestions related to this paper, along with the anonymous reviewers whose observations greatly improved the structure and content of this paper.

References

[Die05] Dietz, P.: The GNU ANSI Common Lisp Test Suite; In *International Lisp Conference Proceedings*, 2005.

[DR04] Dejneka, A. P. and Rhodes, C. S.: Efficient Hardware Arithmetic in Common Lisp; `http://jcsu.jesus.cam.ac.uk/~csr21/papers/modular/modular.pdf`, 2004.

[Gab85] Gabriel, R. P.: *Performance and evaluation of Lisp systems* MIT Press, 1985.

[GKM82] Graham, S. L., Kessler, P. B., and McKusick, M. K.: gprof: a Call Graph Execution Profiler; *ACM SIGPLAN Notices*, 17(6):120–126, 1982.

[KdRB91] Kiczales, G., des Rivières, J., and Bobrow, D. G.: *The Art of the Metaobject Protocol* MIT Press, 1991.

[Khu07] Khuong, P. V.: Implementing an efficient `string= case` in Common Lisp; `http://www.discontinuity.info/~pkhuong/string-case.pdf`, 2007.

[Lev09] Levine, N.: Lisp outside the box; O'Reilly, forthcoming. Draft of chapter 20 at `http://lisp-book.org/contents/ch20.html`, retrieved 2010-01-17, 2009.

[LFM01] Le Fessant, F. and Maranget, L.: Optimizing Pattern Matching; In *ICFP'01 Proceedings*, pages 26–37, 2001.

[Mac92] MacLachlan, R.: CMUCL User's Manual; Technical Report CMU-CS-92-161, Carnegie-Mellon University, 1992 (updated version available at `http://common-lisp.net/project/cmucl/doc/cmu-user/`).

[Nag01] Naggum, E.: `with-hashed-identity`; Usenet group `comp.lang.lisp`, Message-ID `<3208606982556119@naggum.net>`, 2001.

[NR08] Newton, J. and Rhodes, C.: Custom Specializers in Object-Oriented Lisp; *Journal of Universal Computer Science*, 14(20):3370–3388, 2008 `http://www.jucs.org/jucs_14_20/custom_specializers_in_object`.

[PC94] Pitman, K. and Chapman, K., editors *Information Technology – Programming Language – Common Lisp* Number 226–1994 in INCITS. ANSI, 1994.

[Rho04] Rhodes, C.: Grouping Common Lisp Benchmarks; In *1st European Lisp and Scheme Workshop*, Oslo, Norway, 2004.

[Rho06] Rhodes, C.: Revisiting `CONCATENATE-SEQUENCE`; Document 3, Common Lisp Document Repository, 2006 `http://cdr.eurolisp.org/document/3`.

[Rho07] Rhodes, C.: User-extensible Sequences in Common Lisp; In *International Lisp Conference Proceedings*, 2007.

[Riv94] Rivest, R. L.: The RC5 Encryption Algorithm; In *Proceedings of the Second International Workshop on Fast Software Encryption*, pages 86–96. Springer, 1994.

[RR04] Redwine, K. and Ramsey, N.: Widening integer arithmetic; In Duesterwald, E., editor, *Conference on Compiler Construction*, number 2985 in LNCS, pages 232–249. Springer-Verlag, Berlin, 2004.
[SE94] Srivastava, A. and Eustace, A.: ATOM: a system for building customized program analysis tools; In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, 1994.
[Sei05] Seibel, P.: *Practical Common Lisp* Apress, Inc., 2005 `http://gigamonkeys.com/book/`.
[Ste84] Steele, G. L., Jr: *Common Lisp: The Language* Digital Press, 1984.
[SVM04] Strandh, R., Villeneuve, M., and Moore, T.: Flexichain: An editable sequence and its gap-buffer implementation; In *1st European Lisp and Scheme Workshop*, Oslo, Norway, 2004.
[Wat91] Waters, R. C.: Determining the Coverage of a Test Suite; *ACM Lisp Pointers*, 4(4):33–43, 1991.
[Wei09] Weitz, E.: Portable Perl-compatible regular expressions for Common Lisp; `http://weitz.de/cl-ppcre/`, 2009.
[Xi03] Xi, H.: Dependently typed pattern matching; *Journal of Universal Computer Science*, 9(8):851–872, 2003 `http://www.jucs.org/jucs_9_8/dependently_typed_pattern_matching`.

A Implementation of RC5

The code below, in conjunction with the definitions of `rotl32` and `rotr32` from figure 7, implements RC5 encryption and decryption of double words with `encrypt` and `decrypt` as defined in [Riv94], with the key being selected using `setup`. Given a portable definition of the `rotl32` and `rotr32` functions, the code below would continue to work, but any Common Lisp implementation which fails to compile the arithmetic and rotation operations into the equivalent hardware instructions would run the code orders of magnitude more slowly.

```
(defvar *s*)
(declaim (type (simple-array (unsigned-byte 32) (26)) *s*))

(defmacro +/32 (&rest args)
  `(logand (+ ,@args) #xffffffff))
(defmacro -/32 (&rest args)
  `(logand (- ,@args) #xffffffff))

(defun encrypt (p0 p1)
  (declare (type (unsigned-byte 32) p0 p1))
  (do* ((a (+/32 p0 (aref *s* 0))
           (+/32 (rotl32 (logxor a b) b) (aref *s* (* 2 i))))
        (b (+/32 p1 (aref *s* 1))
           (+/32 (rotl32 (logxor b a) a) (aref *s* (1+ (* 2 i)))))
        (i 1 (1+ i)))
       ((> i 12) (values a b))
    (declare (type (unsigned-byte 32) a b))))

(defun decrypt (c0 c1)
  (declare (type (unsigned-byte 32) c0 c1))
  (do* ((b c1 (logxor (rotr32 (-/32 b (aref *s* (1+ (* 2 i)))) a) a))
```

```
        (a c0 (logxor (rotr32 (-/32 a (aref *s* (* 2 i))) b) b))
        (i 12 (1- i)))
       ((<= i 0) (values (-/32 a (aref *s* 0))
                         (-/32 b (aref *s* 1))))
    (declare (type (unsigned-byte 32) c0 c1))))

(defun setup (k)
  (declare (type (simple-array (unsigned-byte 8) (16)) k))
  (let ((l (make-array 4 :element-type '(unsigned-byte 32)))
        (s (make-array 26 :element-type '(unsigned-byte 32))))
    (fill l 0)
    (do ((i 15 (1- i)))
        ((< i 0))
      (setf (aref l (truncate i (/ 32 8)))
            (logior (ash (aref l (truncate i (/ 32 8))) 8) (aref k i))))
    (setf (aref s 0) #xb7e15163)
    (do ((i 1 (1+ i)))
        ((>= i 26))
      (setf (aref s i) (+/32 (aref s (1- i)) #x9e3779b9)))
    (do* ((a 0)
          (b 0)
          (k 0 (1+ k))
          (i 0 (mod (1+ i) 26))
          (j 0 (mod (1+ j) 4)))
         ((>= k (* 3 26)) (setf *s* s))
      (setf (aref s i) (rotl32 (+/32 (aref s i) a b) 3))
      (setf a (aref s i))
      (setf (aref l j) (rotl32 (+/32 (aref l j) a b) (+/32 a b)))
      (setf b (aref l j)))))
```

B Implementation of pathname-type specializers

The code below allows the code in figure 9 to compile and run, dispatching to the relevant method based on the `pathname-type` slot of its argument. The implementation below is not fully general, as the generic function metaclass it defines is only capable of supporting single-argument functions where all specializers must be of the `pathname-type` kind; a more complete implementation, interoperating with other kinds of specializers, requires an adapted protocol for applicable method computation and combination, the subject of further work. The code presented requires the SBCL Metaobject Protocol and (for efficiency) the `string-case` macro from [Khu07].

```
(defclass pathname-type-specializer (specializer)
  ((string :reader pathname-type-specializer-string :initarg :string)
   (direct-methods :initform nil :reader specializer-direct-methods)))
(defvar *pathname-type-specializer-table*
  (make-hash-table :test 'equal))
(defun ensure-pathname-type-specializer (string)
  (or (gethash string *pathname-type-specializer-table*)
      (setf (gethash string *pathname-type-specializer-table*)
            (make-instance 'pathname-type-specializer :string string))))
```

```
(defclass pathname-type-generic-function (standard-generic-function)
  ()
  (:metaclass funcallable-standard-class))
(defmethod compute-discriminating-function
    ((gf pathname-type-generic-function))
  (lambda (pathname)
    (let* ((methods (generic-function-methods gf))
           (function (compiled-discriminating-function methods gf)))
      (set-funcallable-instance-function gf function)
      (funcall function pathname))))

(defun discriminating-function-lambda (methods gf)
  (let (clauses)
    (dolist (m methods)
      (let* ((specializer (car (method-specializers m)))
             (string (pathname-type-specializer-string specializer))
             (form `(funcall ,(method-function m) (list pathname) nil)))
        (push `(,string ,form) clauses)))
    `(lambda (pathname)
       (unless (pathnamep pathname)
         (no-applicable-method ,gf pathname))
       (let* ((type (pathname-type pathname))
              (string (coerce type '(simple-array character (*)))))
         (string-case
             (string :default (no-applicable-method ,gf pathname))
           ,@clauses)))))
(defun compiled-discriminating-function (methods gf)
  (compile nil (discriminating-function-lambda methods gf)))

(defmethod make-method-specializers-form
    ((gf pathname-type-generic-function) method names environment)
  `(list (ensure-pathname-type-specializer ,(cadar names))))

(defmethod add-direct-method
    ((specializer pathname-type-specializer) method)
  (pushnew method (slot-value specializer 'direct-methods)))
(defmethod remove-direct-method
    ((specializer pathname-type-specializer) method)
  (setf (slot-value specializer 'direct-methods)
        (remove method (slot-value specializer 'direct-methods))))

(defmethod unparse-specializer-using-class
    ((gf pathname-type-generic-function)
     (specializer pathname-type-specializer))
  `(pathname-type ,(pathname-type-specializer-string specializer)))
(defmethod parse-specializer-using-class
    ((gf pathname-type-generic-function) name)
  (typecase name
    ((eql t) (find-class t))
    ((cons (eql pathname-type))
     (ensure-pathname-type-specializer (cadr name)))
    (specializer name)))
```