Unportable (but fun)

Using SBCL Internals

Christophe Rhodes

c.rhodes@gold.ac.uk

Goldsmiths, University of London

European Lisp Symposium

28th May 2009

Introduction

Why you should stop worrying and love your implementation:

- there are some neat tools;
- it makes the deliverable possible;
- it's fun.

Also: only through experimenting can we improve on what we currently have.

Plan for this tutorial:

1. developer tools;
2. case studies:
   1. cryptography and hardware arithmetic;
   2. run-time modifiable `string-case`.

Introduction

Why you should stop worrying and love your implementation:

- there are some neat tools;
- it makes the deliverable possible;
- it's fun.

Also: only through experimenting can we improve on what we currently have.

Plan for this tutorial:

1. developer tools;
2. case studies:
    1. cryptography and hardware arithmetic;
    2. run-time modifiable `string-case`.

Developer Tools

Extensions and Contribs

Useful in the course of:

- normal development;
- software archaeology
  - what is the cause of all that allocation?
  - where is all the time being spent?
  - what code is live (and what's dead)?
  - make this code more debuggable!

Extensions

Package Locks

Motivation:

- make it easier to enforce package discipline;
- catch errors in refactoring early.

Package lock behaviour:

- modelled after CLHS, section 11.1.2.1.2;
- restrictions on
  - modifications of *packages* themselves;
  - actions on *symbols* in locked packages.

Extensions

Package Locks

Modifications of packages:

1. shadowing a symbol in a package;
2. importing a symbol to a package;
3. uninterning a symbol from a package;
4. exporting a symbol from a package;
5. unexporting a symbol from a package;
6. changing the packages used by a package;
7. renaming a package;
8. deleting a package.

Extensions

Package Locks

Modifications of symbols:

1. binding or modifying its value;
2. defining or binding it or `(setf it)` as a function;
3. defining or binding it as a macro;
4. defining it as a type specifier or structure;
5. defining it as a `declaration`;
6. declaring or proclaiming it `special`;
7. declaring or proclaiming its `type` or `ftype`;
8. defining a `setf` expander for it;
9. defining it as a `method-combination` type;
10. using it as the `class-name` argument to `setf` of `find-class`.

Extensions

Package Locks

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

(in-package "FOO")

(defun frob () ...)
```

Extensions

Package Locks

```
(defpackage "FOO"
  (:export "FROB" "FROB-POP" "WITH-FROB-POP")
  (:lock t)
  (:implement))

(defpackage "FOO-INT"
  (:use "CL" "SB-EXT")
  (:implement "FOO" "FOO-INT"))

(in-package "FOO-INT")

(defun frob () ...)
```

Extensions

Package Locks

A catch – local redefinitions:

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

Package-lock-friendly version:

```
(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))))
```

Extensions

Package Locks

A catch – local redefinitions:

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

Package-lock-friendly version:

```
(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))))
```

Extensions

Compiler Policy

Basic policy symbols as standardized:

- `speed`, `space`, `safety`, `debug`, `compilation-speed`;

Finer-grained policies taken from main policies:

- `merge-tail-calls`;
- `preserve-single-use-debug-variables`;
- `insert-debug-catch`;
- ... and more.

Finer-grained policies are overrideable:

```
(declaim (optimize sb-c::merge-tail-calls))
```

Extensions

Compiler Policy

`restrict-compiler-policy` operator:

- intended for interactive use;
- defines minimum values for compiler policies.

Use cases:

- why does this ancient body of code segfault?
  - `(restrict-compiler-policy 'safety 3)`
- why is this (huge) function going wrong?
  - `(restrict-compiler-policy 'debug 3)`
  - `C-u C-c C-c` in SLIME.

Contribs

sb-sprof

Statistical profiler – basic idea:

- periodically interrupt the running program;
- acquire information about the state;
- finally report accumulated information.

Less-known information:

- not just cpu-time: `:mode` argument:
  - `:time` provides wall-clock timing;
  - `:alloc` provides allocation profiling.
- includes call-counting (lightweight deterministic profiling);
- disassembler integration.

Contribs

sb-sprof

```
(defun sb-sprof-example-fun (x y)
  (declare #+(or) (type fixnum x)
           (type (unsigned-byte 16) y))
  (dotimes (i y)
    ;; exercise: see what happens when you replace
    ;; the quotient with (1+ most-positive-fixnum)
    (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))))
```

Contribs

sb-sprof

Output for :cpu mode:

```
           Self        Total        Cumul
  Nr  Count     %  Count     %  Count     %    Calls  Function
------------------------------------------------------------------------
   1    334  33.4    334  33.4    334  33.4        -  TRUNCATE
   2    294  29.4    773  77.3    628  62.8        -  SB-SPROF-EXAMPLE-FUN
   3    189  18.9    220  22.0    817  81.7        -  SB-VM::GENERIC-+
   4     99   9.9    120  12.0    916  91.6        -  SB-BIGNUM:BIGNUM-TRUNCATE
   5     25   2.5     25   2.5    941  94.1        -  SB-BIGNUM::%NORMALIZE-BIGNUM
   6     24   2.4     30   3.0    965  96.5        -  SB-KERNEL:TWO-ARG-<
   7      9   0.9      9   0.9    974  97.4        -  SB-BIGNUM:BIGNUM-PLUS-P
   8      0   0.0    998  99.8    974  97.4        -  SB-SPROF-EXAMPLE
```

Contribs

sb-sprof

Output for `:time` mode:

```
           Self        Total        Cumul
  Nr  Count     %  Count     %  Count     %    Calls  Function
------------------------------------------------------------------------
   1     83   8.3     83   8.3     83   8.3        -  TRUNCATE
   2     68   6.8    937  93.7    151  15.1        -  SB-SPROF-EXAMPLE-FUN
   3     52   5.2     61   6.1    203  20.3        -  SB-VM::GENERIC-+
   4     25   2.5     26   2.6    228  22.8        -  SB-BIGNUM:BIGNUM-TRUNCATE
   5      5   0.5      5   0.5    233  23.3        -  SB-BIGNUM:BIGNUM-PLUS-P
   6      4   0.4      9   0.9    237  23.7        -  SB-KERNEL:TWO-ARG-<
   7      1   0.1      1   0.1    238  23.8        -  SB-BIGNUM::%NORMALIZE-BIGNUM
   8      0   0.0   1000 100.0    238  23.8        -  SB-SPROF-EXAMPLE
[...]
  38      0   0.0    755  75.5    238  23.8        -  SLEEP
------------------------------------------------------------------------
            762  76.2                                     elsewhere
```

Contribs

sb-sprof

Output for :alloc mode:

```
           Self        Total        Cumul
  Nr  Count     %  Count     %  Count     %    Calls  Function
-------------------------------------------------------------------------
   1    886  88.6    886  88.6    886  88.6         -  SB-VM::GENERIC-+
   2    107  10.7    107  10.7    993  99.3         -  TRUNCATE
   3      5   0.5      5   0.5    998  99.8         -  SB-BIGNUM:BIGNUM-TRUNCATE
   4      0   0.0   1000 100.0    998  99.8         -  SB-SPROF-EXAMPLE
```

Contribs

sb-cover

Code coverage tool – basic idea:

- associate code with markers;
- insert code to frob marker after executing code;
- interrogate state of coverage data;
- generate pretty html reports.

Particularly useful when:

- writing a test suite;
- investigating code paths for a particular workload.

Contribs

sb-cover

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

Then browse `#u"file:///tmp/cl-ppcre/cover-index.html"`.

Case study I: RC5 Encryption
(Rivest, 1997)

Design goals of RC5:

- symmetric block cipher;
- fast, word-oriented;
- adaptable;
- simple;
- high security;

Case study I: RC5 Encryption

Modular Arithmetic

Close to the metal?

- Lisp integers are unbounded;
  - no silent wrongness;
  - implemented in software.
- Hardware (usually) supports fixed-width integers
  - arithmetic performed in $\mathbb{Z}_{2^{32}}$;
  - fast;
  - differently correct.

Case study I: RC5 Encryption

Modular Arithmetic

How to recover speed and correctness?

- request arithmetic in $\mathbb{Z}_{2^{32}}$ explicitly;
- `(logand` *expression* `#xffffffff)`;
- SBCL automatically translates generic arithmetic in *expression* to equivalent modular form;
- modular arithmetic is then compiled to small sequences of machine instructions.

Case study I: RC5 Encryption

Modular Arithmetic

‘Modular arithmetic’

- recognized and performed automatically;
- speed declarations not necessary
  - `(unsigned-byte 32)` type declarations helpful;
  - 64-bit modular arithmetic on x86-64 and alpha.
- signed-arithmetic variant is harder to express
  - no non-conditional idiom in portable CL;
  - use `sb-c::mask-signed-field` instead.

Case study I: RC5 Encryption

Bitwise rotation and compiler support

Bitwise rotation:

- ‘C’ notation: `((x << y) | (x >> (32-y)));`
- three instructions where one will do, even with modular arithmetic.

Make a rotation function known to the compiler:

```
(sb-vm::defknown %rotr
      ((unsigned-byte 32) (unsigned-byte 5))
    (unsigned-byte 32)
  (sb-c::foldable sb-c::flushable sb-c::movable))
```

Case study I: RC5 Encryption

Bitwise rotation and compiler support

Bitwise rotation:

- ‘C’ notation: `((x << y) | (x >> (32-y)));`
- three instructions where one will do, even with modular arithmetic.

Make a rotation function known to the compiler:

```
(sb-vm::defknown %rotr
      ((unsigned-byte 32) (unsigned-byte 5))
    (unsigned-byte 32)
  (sb-c::foldable sb-c::flushable sb-c::movable))
```

Case study I: RC5 Encryption

Bitwise rotation and compiler support

Now make the compiler know how to compile `%rotr` efficiently:

```
(sb-vm::define-vop (%rotr)
  (:policy :fast-safe)
  (:translate %rotr)
  (:note "inline 32-bit rotation")
  (:args (integer :scs (sb-vm::unsigned-reg))
         (count :scs (sb-vm::unsigned-reg) :target ecx))
  (:arg-types sb-vm::unsigned-num sb-vm::unsigned-num)
  (:temporary (:sc sb-vm::unsigned-reg :offset sb-vm::ecx-offset)
              ecx)
  (:results (res :scs (sb-vm::unsigned-reg)))
  (:result-types sb-vm::unsigned-num)
  (:generator 5
     (sb-vm::move res integer)
     (sb-vm::move ecx count)
     (sb-vm::inst sb-vm::ror res :cl)))
```

Case study II: modifiable string-case

A contrived example

A contrived example:

- elements of the MOP:
  - because no CL tutorial is complete without mention of the MOP;
  - steering clear of *de-facto* portable bits.
- portable string pattern-matching...
- backed up by unportable efficiency tweaks.

Basic idea:

- assume logfile lines of the form "*prefixid*:*rest of line*";
- dispatch to particular code based on *prefixid*

Case study II: modifiable string-case

A contrived example

A contrived example:

- elements of the MOP:
  - because no CL tutorial is complete without mention of the MOP;
  - steering clear of *de-facto* portable bits.
- portable string pattern-matching...
- backed up by unportable efficiency tweaks.

Basic idea:

- assume logfile lines of the form "*prefixid*:*rest of line*";
- dispatch to particular code based on *prefixid*

Case study II: modifiable string-case

A contrived example

```
(defun frob (prefix rest)
  (cond
    ((string= prefix "httpd") ...)
    ((string= prefix "exim") ...)
    ((string= prefix "atd") ...)
    ((string= prefix "ntpd") ...)
    (t (warn "unrecognized: ~S" prefix))))
```

Characteristics:

- ugly;
- hard to modify;
- inefficient.

Case study II: modifiable string-case

`string-case` itself

```
(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)))))))
```

Characteristics:

- not so ugly;
- hard to modify;
- inefficient.

Case study II: modifiable string-case

Efficient `string-case`

`string-case` knows the strings it's after at compile time.

- suggests pattern-matching approach;
- build search tree, using O(1) string access;
- strings are equal if `logior` of `logxor` of `char-codes` is 0;
- tune balance between branches and extra work;
- P.Khuong, *Implementing an efficient `string=` case in Common Lisp*, 2008

Characteristics:

- not so ugly;
- hard to modify;
- efficient.

Case study II: modifiable string-case

`string-case` and generic functions

The final piece: aim to write code like

```
(defgeneric frob (prefix rest)
  (:generic-function-class magic-generic-function))

(defmethod frob ((prefix (string= "httpd")) rest)
  ...)
(defmethod frob ((prefix (string= "exim")) rest)
  ...)
```

while

- preserving the efficiency that has been gained;
- allowing arbitrary addition and removal of methods.

Case study II: modifiable string-case

string-case and generic functions

The final piece: aim to write code like

```
(defgeneric frob (prefix rest)
  (:generic-function-class magic-generic-function))

(defmethod frob ((prefix (string= "httpd")) rest)
  ...)
(defmethod frob ((prefix (string= "exim")) rest)
  ...)
```

while

- preserving the efficiency that has been gained;
- allowing arbitrary addition and removal of methods.

Case study II: modifiable string-case

`string-case` and generic-functions

Ingredients:

1. new generic function class `magic-generic-function`;
2. new specializer class `string=-specializer`;
3. new method on `compute-discriminating-function`;
4. new method on `make-method-specializers-form`;
5. bookkeeping methods on `add-direct-method` and `remove-direct-method`;
6. (optional) runtime methods to help `find-method` and `print-object`.

Characteristics:

- not ugly at all;
- easy to modify and factor appropriately;
- efficient.

Conclusions

Unportability is fun! (and can be productive). And there's more...

- stepper;
- dynamic-extent declarations;
- compare-and-swap support;
- hooking into type derivation;
- generic sequences;
- customizing the FFI;
- ... and things I don't know about.

Acknowledgments

Alexey Dejneka, Paul Khuong, David Lichteblau, Jim Newton, Nikodemus Siivola, Juho Snellman

The SBCL community