The GCL ANSI Common Lisp Test Suite

Paul Dietz
Motorola Global Software Group

Outline of Talk

- Goals
- Conformance tests
- Specialized testers
- Comments on X3.226

Goals

Primary goal:

> Produce a tool for assisting implementors in achieving and maintaining compliance with the ANSI CL standard.

Secondary goals:

- Familiarize myself with the CL standard
- Explore testing methods
- Test the standard itself

Non-Goals

- Measuring compliance
- Ranking implementations by compliance
- Changing the CL standard

Sources

- Harlequin/Lispworks Common Lisp Hyperspec – derived from the ANS X3.226 standard
- Feedback from implementors
- Discussions on comp.lang.lisp, email

Implementations Tested

- Allegro CL (6.2 and 7.0; x86, Sparc, Power)
- Armed Bear Common Lisp (ABCL) (JVM on x86)
- CLISP (x86)
- CMU CL (x86)
- ECL (x86)
- GNU Common Lisp (GCL) (x86, other Debian platforms)
- Lispworks 4.* (x86)
- Open MCL (Power)
- Steel Bank CL (x86, Sparc, Power, Alpha, MIPS)

Implementations Not Tested

- Symbolics
- Liquid Common Lisp
- Xerox Common Lisp
- WCL
- Corman Common Lisp
- Scieneer Common Lisp
- Sacla

Waters' RT package

(deftest *name form expected-values...*)

```
(deftest plus.1 (+) 0)

(deftest flet.4
  (block %f
    (flet ((%f (&optional (x (return-from %f :good)))
               nil))
      (%f)
      :bad))
  :good)
```

```
> (do-test 'decf.order.4)
Test DECF.ORDER.4 failed
Form: (LET ((X 0))
        (PROGN "See CLtS 5.1.3"
           (VALUES (DECF X (SETF X 1)) X)))
Expected values: 0
                 0
Actual values: -1
               -1.
NIL
```

Changes to RT

- Optionally catch errors (treat as failure)
- Optionally compile forms
- Expected results compared with `EQUALP-WITH-CASE`
- Test annotation
- Expected failures
- $O(n)$ time, $n =$ number of tests

Section	Size	Tests	Section	Size	Tests
Arrays	212623	1109	Pathnames	47100	215
Characters	38655	256	Printer	454314	2364
Conditions	71250	658	Reader	101662	663
Cons	264208	1816	Sequences	562210	3219
D/C Flow	185973	1217	Streams	165956	796
Environment	51110	206	Strings	83982	415
Eval/Compile	41638	234	Structures	46271	1366
Files	26375	87	Symbols	106063	1141
Hash Tables	38752	158	System Cons.	16909	77
Iteration	98339	767	Types	104804	599
Numbers	290991	1382	Misc	291883	679
Objects	283549	774	Infrastructure	115090	
Packages	162203	493	Random	190575	

Example of Bugs Found: SBCL

Found 219 bugs, fixed in releases 0.7.8 to 0.9.0

Kind	Bugs	Kind	Bugs	Kind	Bugs
Type Inf	13	Symbols	1	Hash Tables	3
Compiler	20	Packages	3	Filenames	3
Reader	5	Numbers	17	Files	4
Types	16	Characters	0	Streams	17
Eval/DCF	15	Cons	10	Printer	29
Iteration	10	Arrays	8	Sys. Con.	1
Objects	26	Strings	0	Conditions	8
Structures	4	Sequences	6	Environment	0

Testing Strategies

- Simple tests.
  Most tests in suite are of this kind.
- Exhaustive tests.
  Confirm some predicate applies to all elements of some large set.
- Randomized tests.
  Evade combinatorial explosion by random sampling of a test space.

Common idiom: confirm some property holds for ‘all’ lisp values

```
(deftest sxhash.1
  (loop for x in *universe*
        for hash-code = (sxhash x)
        unless (typep hash-code '(and unsigned-byte fixnum))
        collect x)
  nil)
```

This test found a bug in SBCL!

Randomized Tests

Myers (in *The Art of Software Testing*):

> "Probably the poorest ... methodology of all."

Others have had good results:

- Miller's 'fuzz testing'
- McKeeman (C compilers)
- Slutz (SQL systems)
- Lindig (C procedure calls)

Objections to Randomized Testing

- Inefficient
  - Optimizes test creation vs. test execution
- Irreproducible
  - Common bugs recur anyway
  - Properly designed tests report failing inputs

Objections to Randomized Testing (cont.)

- Ignores knowledge of program being tested.
  - Knowledge may not be available (black box testing)
  - May be wrong or misleading
  - Semi-random tests can incorporate knowledge

Randomized Tests (continued)

- Tests of functions with many keyword arguments
- Print/read consistency of random objects
- Random math operands
- Subtypes
- Compiler tests

Print/Read Consistency

- Bind printer control variables to random values.
- Bind `*PRINT-READABLY*` to true.
- Print random objects, read again.
- Confirm that objects are ‘similar’.

Subtypes

- Generate random types T_1, T_2.

- If $T_1 \subseteq T_2$ and `SUBTYPEP` succeeds, check:

$$\overline{T_2} \subseteq \overline{T_1}$$
$$T_1 \cap \overline{T_2} \subseteq \emptyset$$
$$T \subseteq \overline{T_1} \cup T_2$$

- If $T_1 \not\subseteq T_2$ and `SUBTYPEP` succeeds, check:

$$\overline{T_2} \not\subseteq \overline{T_1}$$

```
(let ((t1 '(not (not t)))
      (t2 '(or rational t)))
  (values
    (multiple-value-list (subtypep t1 t2))
    (multiple-value-list (subtypep `(and ,t1 (not ,t2)) nil))))
==>
(T T)
(NIL T)
```

Compiler Testing

Behavior-preserving transformations are opportunities for random testing.

- Type declarations
- `THE` forms
- `OPTIMIZE` settings
- `INLINE` and `NOTINLINE`
- `EVAL` vs. `COMPILE`

Tests of Type Propagation/Inference

- Type inference very useful for efficient lisp compilation.
  - Unboxing
  - Elimination of runtime dispatch
  - Folding runtime type checks, bounds checks
- Not well tested by usual tests in suite

Testing of Type Propagation (continued)

Strategy:

- For some function F, generate random arguments $x_1, \ldots, x_k$.
- `EVAL` $(Fx_1 \ldots x_k)$.
- Generate a lambda form with:
  - Some subset of the parameters as formal parameters
  - Random optimize levels
  - Random declaration of formal parameter types
  - Random `THE` forms.
- Compile, apply, and compare results.

```
(def-type-prop-test |+.1| '+ '(integer integer) 2)

;;; Form: +
;;; Parameters:  -635 -221
;;; Lambda form:
    (lambda (p1)
      (declare (optimize (speed 1) (safety 1) (debug 1) (space 2))
         (type (integer -4730 9617) p1))
      (+ (the (integer * 862277) p1) -221))
```

```
(funcall (compile nil
    '(lambda () (declare (optimize debug)) (symbolp -86755))))
  ==> segmentation violation


(compile nil '(lambda (x) (declare (type (member 4 -1) x)
                                   (optimize speed (safety 1)))
                    (isqrt x)))
  ==> "Error: -1 is illegal argument to isqrt"
```

```
(compile nil '(lambda (p1)
                (declare (optimize (speed 1) (safety 2)
                                   (debug 2) (space 0))
                         (type keyword p1))
                (keywordp p1)))
   ==> failed AVER: "(EQ CHECK SIMPLE)"
```

Random Compiler Stress Tester

- Generate random integer-valued form with integer arguments.
- Wrap in two lambda forms
  - One with type declarations, the other with none.
  - One has all Common Lisp functions declared NOTINLINE.
- Compile and apply one, eval the other.
- Are results the same?

- Found bugs within seconds in all implementations.
- Most failures were assertion failures, type errors, or incorrect values.
- Bugs that crashed the lisp were infrequent.
- Many dead code, type inference bugs.

```
(funcall
   (compile nil '(lambda (b)
                  (declare (type (integer 8 22337) b))
                  (+ b 2607688420)))
   100)  ==> incorrect value

(funcall (compile nil
   '(lambda () (flet ((%f12 () (unwind-protect 1))) 0))))
==> "The value NIL is not of type SB-C::NODE."

(labels ((%f17 (f17-1 f17-2
                &optional (f17-3 (unwind-protect 178)))
               483633925))
  -661328075)
==> "The assertion (EQ (C::COMPONENT-KIND C:COMPONENT) :INITIAL)
failed."
```

Experience with CLISP

- Total of 14 compiler bugs found in CLISP by this tester.
- No current failures (except for bignum overflow).
- $\approx$ 200 million random tests were run.

Automated Pruning

- Forms produced by the random compiler tester can be very large.
- Pruner simplifies them to minimal forms, preserving failure.
- Minimal forms are usually small (but not always!)
- Pruner limits random forms.

To do: improve the pruner so more forms can be tested.

Comments on the Standard

- Some things were difficult to test.
  - Too much freedom for the implementation (pathnames).
  - Not well specified (floating point accuracy).
  - Ambiguities.
- Unintended consequences:
  - Type upgrading
  - `TYPE-OF`

```
(TYPE-OF 17) ==> FIXNUM
```

Is this compliant with the standard?

No!

> "For any object that is an element of some built-in type [...] the type returned is a recognizable subtype of that built-in type."
>
> built-in type n. one of the types in Figure 4-2.

Figure 4.2 contains the type `UNSIGNED-BYTE`, which contains 17, but is not a supertype of `FIXNUM`.

A Problem With `UPGRADED-ARRAY-ELEMENT-TYPE`

"A type is always a subtype of its upgraded array element type. Also, if a type T_x is a subtype of another type T_y, then the upgraded array element type of T_x must be a subtype of the upgraded array element type of T_y." (section 15.1.2.1)

This implies:

If T_z is the intersection of T_x and T_y, then
(`U-A-E-T` T_z) is equivalent to (`U-A-E-T` T_x) $\cap$ (`U-A-E-T` T_y).

- If `(UNSIGNED-BYTE 8)` and `(SIGNED-BYTE 8)` are specialized array element types, then so must be `(UNSIGNED-BYTE 7)`.
  - SBCL required the addition of three more specialized integer array element types.
- Since `BIT` and `CHARACTER` are specialized array element types, then so must be `NIL`.
  - A conforming lisp must have arrays specialized to hold *nothing*!?
- Vectors of `NIL`-type are strings!

Future Work

- Complete the test suite
- Extend random compiler tester to more of Common Lisp
- Random testing of CLOS
- Test non-ANSI behaviors

Questions?