Parenthetically Speaking with Kent M. Pitman

The Best of Intentions

EQUAL Rights—and Wrongs—in Lisp

Operations in Lisp, Scheme, and other dynamically-typed languages typically dispatch on representational type information rather than intentional type information. Several broad classes of bugs and confusions can be traced to improper attempts to recover intentional type information from representation types.

I've chosen here to discuss some Common Lisp built-in operators that highlight the various issues. However, the problems cited here are quite general, and occur routinely in other dynamically-typed languages as well as user programs. Fortunately, the solutions to these problems are also conveniently available to designers, implementors, and programmers—without throwing dynamic typing out the window. I've provided code to illustrate how these ideas can be translated into practice without requiring any fundamental change to the underlying technology.

Copying

"Why is there no generic `COPY` function?" Common Lisp programmers often ask this question.

This glossary entry from *dpANS Common Lisp* [CL93] provides some useful background information and a brief rationale for the absence of a generic `COPY` function:

> **copy** *n.*
>
> 1. (of a *cons C*) a *fresh cons* with the *same car* and *cdr* as *C*.
> 2. (of a *list L*) a *fresh list* with the *same elements* as *L*.. (Only the *list structure* is *fresh*; the *elements* are the *same*.) See the function `copy-list`.
> 3. (of an *association list A* with *elements* A_i) a *fresh list B* with *elements* B_i, each of which is `nil` if A_i is `nil`, or else a *copy* of the *cons* A_i. See the function `copy-alist`.
> 4. (of a *tree T*) a *fresh tree* with the *same leaves* as *T*. See the function `copy-tree`.
> 5. (of a *random state R*) a *fresh random state* that, if used as an argument to the *function* random would produce the same series of "random" values as *R* would produce.
> 6. (of a *structure S*) a *fresh structure* that has the same *type* as *S*, and that has slot values, each of which is the *same* as the corresponding slot value of *S*.
>
> (Note that since the difference between a *cons*, a *list*, and a *tree* is a matter of "view" or "intention," there can be no general-purpose *function* which, based solely on the *type* of an *object*, can determine which of these distinct meanings is intended. The distinction rests solely on the basis of the text description within this document. For example, phrases like "a *copy* of the given *list*" or "copy of the *list x*" imply the second definition.)

Parenthetically Speaking expresses opinions and analysis about the Lisp family of languages. Except as explicitly indicated otherwise, the opinions expressed are those of the author and do not necessarily reflect the official positions of any organization or company with which the author is affiliated. Kent M. Pitman can be reached via the Internet as KMP@Harlequin.COM, or by U.S. mail at Harlequin, Inc., Suite 904, One Cambridge Center, Cambridge, MA 02142 U.S.A.

Programmers want a generic COPY function that somehow "does the right thing," but relying solely on the representation type is insufficient to resolve intentional distinctions between conses, lists, alists, and trees, so we wouldn't know whether to call COPY-CONS (if there were such a thing), COPY-LIST, COPY-ALIST, or COPY-TREE.

Of course, we could just define that COPY-LIST is what's used, but that could (and probably would) give the misleading impression that such a choice was "uniquely determined" or even "morally correct," instead of just arbitrarily chosen. Also, an arbitrary choice would leave users feeling funny about the presence of an operator that can copy many—but not all—kinds of objects.

Where does this leave us? Well, it doesn't mean the problem cannot be solved. It only means that additional data flow must be provided to the running program in order for it to be able to divine the programmer's intent. For example:

```
(DEFUN COPY (OBJECT &OPTIONAL (INTENTIONAL-TYPE (TYPE-OF OBJECT)))
  (COPY-OBJECT OBJECT INTENTIONAL-TYPE))
(DEFGENERIC COPY-OBJECT (OBJECT INTENTIONAL-TYPE))
(DEFMETHOD COPY-OBJECT ((OBJECT CONS) (TYPE (EQL 'CONS)))
  (CONS (CAR OBJECT) (CDR OBJECT)))
(DEFMETHOD COPY-OBJECT ((OBJECT LIST) (TYPE (EQL 'LIST)))
  (COPY-LIST OBJECT))
(DEFMETHOD COPY-OBJECT ((OBJECT LIST) (TYPE (EQL 'ALIST)))
  (COPY-ALIST OBJECT))
(DEFMETHOD COPY-OBJECT (OBJECT (TYPE (EQL 'CONS-TREE)))
  (COPY-TREE OBJECT))
```

Equality

"If I can't have COPY, why can I have a generic (ok, just polymorphic) EQUAL function?" Common Lisp programmers *should* ask this question, but rarely do. I suppose most programmers are just happy we've given them a tool of 'reasonable engineering quality' so they can get work done, and are not philosophically inclined to look such a proverbial gift horse in the mouth.

The design issues here are pretty much the same as they are for COPY. If COPY can't be done properly, then neither can EQUAL. And, in fact, that's the case. There is no uniquely determined equality function for complex structures—there are only arbitrary ones.

EQUAL and EQUALP are just two of an arbitrary number of possible equality operations that could have been provided by the language. Indeed, many of the dialects which contributed to the design of Common Lisp had functions called EQUAL which had slightly varying semantics. No particular definition was definitively better than another. Arbitrary choices were made to resolve the differences.

To avoid an arbitrary choice, it would be necessary to express some intentional information about how to descend the tree and how to compare the leaves. The following code illustrates one possibility.

```
;;; The name EQUIVALENT is used here instead of EQUAL to avoid a need
;;; to shadow the Common Lisp built-in function EQUAL.
(DEFUN EQUIVALENT (X Y &OPTIONAL (INTENTIONAL-TYPE (TYPE-OF Y)))
  (EQUIVALENT-OBJECTS X Y INTENTIONAL-TYPE))
(DEFGENERIC EQUIVALENT-OBJECTS (X Y INTENTIONAL-TYPE))
(DEFMETHOD EQUIVALENT-OBJECTS ((X CONS) (Y CONS) (TYPE (EQL 'CONS)))
  (AND (EQ (CAR X) (CAR Y)) (EQ (CDR X) (CDR Y))))
(DEFMETHOD EQUIVALENT-OBJECTS ((X LIST) (Y LIST) (TYPE (EQL 'LIST)))
  (AND (= (LENGTH X) (LENGTH Y))
       (EVERY #'EQL X Y)))
(DEFMETHOD EQUIVALENT-OBJECTS ((X LIST) (Y LIST) (TYPE (EQL 'ALIST)))
  (AND (= (LENGTH X) (LENGTH Y))
       (EVERY #'(LAMBDA (X Y) (EQUIVALENT X Y 'CONS)) X Y)))
(DEFMETHOD EQUIVALENT-OBJECTS (X Y (TYPE (EQL 'CONS-TREE)))
  (EQUAL X Y))
```

This example shows truth values under our new EQUIVALENT predicate after various kinds of copying using our new COPY function:

```
(LET* ((A (GENSYM "A")) (B (GENSYM "B"))
       (CONS (LIST A B))
       (LIST (LIST CONS CONS))
       (TYPES '(CONS LIST ALIST CONS-TREE))
       (LISTS (CONS LIST (MAPCAR #'(LAMBDA (TYPE) (COPY LIST TYPE)) TYPES)))
       (PAIRS (LOOP FOR P1 ON LISTS
                    APPEND (LOOP FOR P2 ON (CDR P1)
                                 COLLECT (LIST (CAR P1) (CAR P2))))))
  (LOOP FOR TYPE IN TYPES
        COLLECT (CONS TYPE (LOOP FOR (X Y) IN PAIRS
                                 COLLECT (EQUIVALENT X Y TYPE)))))

   →  ((CONS      T NIL NIL NIL NIL NIL NIL NIL NIL NIL)
       (LIST      T T   NIL NIL T   NIL NIL NIL NIL NIL)
       (ALIST     T T   T   NIL T   T   NIL T   NIL NIL)
       (CONS-TREE T T   T   T   T   T   T   T   T   T))
```

An interesting sidelight on the equality issue is that the functions provided by Common Lisp are not chosen in a completely arbitrary way. The following implications hold:

```
(EQ X Y) ⇒ (EQL X Y) ⇒ (EQUAL X Y) ⇒ (EQUALP X Y)
```

Similarly, in the replacement functionality we've proposed here:

```
(EQUIVALENT X Y 'CONS)
  ⇒ (EQUIVALENT X Y 'LIST)
      ⇒ (EQUIVALENT X Y 'ALIST)
          ⇒ (EQUIVALENT X Y 'CONS-TREE)
```

The fact that these functions have been chosen to have a sort of inclusion relationship does not imply that equality is a one-dimensional quantity, with predicates varying in no more interesting way than being more or less conservative. For example, the following predicates are not comparable:

```
(DEFUN EQUIV1 (X Y)
  (COND ((AND (STRINGP X) (STRINGP Y)) (STRING-EQUAL X Y)) ;Liberal
        ((AND (NUMBERP X) (NUMBERP Y)) (EQL X Y))          ;Conservative
        (T (EQL X Y))))

(DEFUN EQUIV2 (X Y)
  (COND ((AND (STRINGP X) (STRINGP Y)) (STRING= X Y)) ;Conservative
        ((AND (NUMBERP X) (NUMBERP Y)) (= X Y))       ;Liberal
        (T (EQL X Y))))

(EQUIV1 "Foo" "FOO") → T         (EQUIV2 "Foo" "FOO") → NIL
(EQUIV1 1 1.0)       → NIL       (EQUIV2 1 1.0)       → T
```

While it's useful that there is an inclusion relationship among the particular equality predicates offered by Common Lisp, the orderliness of this relationship contributes to a mistaken impression among some programmers that the equality testing done by EQUAL and EQUALP is somehow more special than many similar predicates we could have provided but did not. This is evidenced in occasional bug reports that vendors receive, arguing that an incorrect choice has "clearly" been made for how objects of a given type are compared, rather than acknowledging that the choice is really quite arbitrary. When urged to write their own equality predicate to suit their particular needs, they sometimes react as if we are putting them off, rather than realizing that any function they could write is just as valid as any one the language provides. Were the language changed to accommodate such bug reports, different users would probably complain. EQUAL and EQUALP are not in Common Lisp because they are uniquely dictated by science—rather, these functions are present out of a sense of tradition and conceptual (although in some cases not functional) compatibility with older dialects.

Coercion

Another place where intentional type information is quite valuable is in coercion. The Common Lisp functions STRING and COERCE blatantly illustrate the problem:

```
(STRING 'NIL) → "NIL"          (COERCE 'NIL 'STRING) → ""
```

The issue here is that COERCE coerces sequence types to other sequence types, but does not coerce symbols to strings; STRING, by contrast, accepts only symbols, strings, and characters as arguments. These somewhat arbitrary type restrictions on the arguments provide enough intentional type information to dictate the behavior in the NIL case.

```
(STRING 'FOO) → "FOO"          (COERCE '(#\F #\O #\O) 'STRING) → "FOO"
(STRING 'NIL) → "NIL"          (COERCE '() 'STRING)            → ""
```

An arbitrary design decision was made that COERCE would view NIL as an empty list, but STRING would view it as a symbol. The problem could be solved by a single function in a general way if COERCE took arguments specifying not only the target type but the intentional type of the argument, as in the following example:

```
;;; We use the name CONVERT here to avoid name conflict with CL's COERCE.
(DEFGENERIC CONVERT (X FROM-INTENTIONAL-TYPE TO-INTENTIONAL-TYPE))
(DEFMETHOD CONVERT ((OBJECT SYMBOL) (FROM (EQL 'SYMBOL)) (TO (EQL 'STRING)))
  (STRING OBJECT))
(DEFMETHOD CONVERT ((OBJECT LIST) (FROM (EQL 'LIST)) (TO (EQL 'STRING)))
  (COERCE OBJECT 'STRING))

(CONVERT 'NIL 'SYMBOL 'STRING) → "NIL"
(CONVERT 'NIL 'LIST   'STRING) → ""
```

A related problem arose with the function INT-CHAR, which was removed between the publication of *Common Lisp: The Language* [CL84] and *dpANS Common Lisp* [CL93], primarily because the mapping from integers to characters is not uniquely determined and it was felt that typical uses of INT-CHAR were suspect in portable code because the integer-to-character mapping performed by INT-CHAR is implementation-defined, and the intentional type information of the integer was not manifest. Explicit representation of intentional types would have solved this problem as well, as illustrated here:

```
;;; For brevity, these examples do not bounds-check their argument integers.
(DEFMETHOD CONVERT ((I INTEGER) (FROM (EQL 'ASCII-CODE)) (TO (EQL 'CHARACTER)))
  (AREF *ASCII-CODE-TABLE* I))
(DEFMETHOD CONVERT ((I INTEGER) (FROM (EQL 'EBCDIC-CODE)) (TO (EQL 'CHARACTER)))
  (AREF *EBCDIC-CODE-TABLE* I))
(DEFMETHOD CONVERT ((I INTEGER) (FROM (EQL 'SAIL-CODE)) (TO (EQL 'CHARACTER)))
  ;; This character encoding originated long ago at the Stanford AI Lab.
  (AREF *SAIL-CODE-TABLE* I))

(CONVERT #x41 'ASCII-CODE  'CHARACTER) → #\A
(CONVERT #xC1 'EBCDIC-CODE 'CHARACTER) → #\A

(CONVERT #x08 'ASCII-CODE  'CHARACTER) → #\BS
(CONVERT #x08 'SAIL-CODE   'CHARACTER) → #\λ
```

Input and Output

An example of a programming system that *does* make good use of intentional type information is the Common Lisp Interface Manager [CLIM92].

CLIM has a model of both input and output which is built around the idea of intentional types. Input and output requests can be accompanied by "presentation types" that contains the intentional type information necessary to accept (*i.e.*, parse) or present (*i.e.*, unparse or display) an object in a manner more refined than representational type information would permit. This is especially important for input since there are a wide variety of possible representations into which the same string could be mapped.

```
(ACCEPT-FROM-STRING 'STRING "3.2")    → "3.2"
(ACCEPT-FROM-STRING 'NUMBER "3.2")    → 3.2
(ACCEPT-FROM-STRING 'KEYWORD "3.2")   → :|3.2|
(ACCEPT-FROM-STRING 'PATHNAME "3.2") → #P"MY-HOST:3.2"
```

Note that the presence of this CLIM functionality does not compromise access to pre-existing functionality involving the standard Lisp parser (i.e., READ), as in:

```
(READ-FROM-STRING "3.2") → 3.2
```

Instead, CLIM provides such functionality through one of the many explicitly provided "presentation type" options, as in:

```
(ACCEPT-FROM-STRING 'EXPRESSION "3.2") → 3.2
```

Translating between Internal and External Representations

In fact, input and output are just a specific instance of the more general issue of translating between internal and external representations of any kind. Other instances might occur in binary file I/O, in network protocols, in foreign function interfaces, and even in some cross-module exchanges within the same address space. Simplified, stylized, compacted, or otherwise specialized representations may be quite useful, but are really only fully powerful if the information that they convey can be inverted to produce an object of the same quality as the original.

Conclusions

In a language with strong static typing, the intentional type of the object would be evident at compile time, and the same representational type could be used for multiple intentional types. The main problem with this approach is that it gives up dynamic typing, which Lisp users have come to expect and enjoy.

If static type information is optional, it is difficult for language designers to reliably express how operators behave in the hybrid environment that results.

To avoid the need to throw dynamic typing out the window, we have proposed that the information which in some languages would be reliably available at compile time be passed as explicit data. Where such data is provided as a literal constant, the same compilation techniques as used by static languages would still apply. Where such data is not available to the compiler, the information would still be reliably and explainably available at runtime.

We have seen how this technique would lead to a more intuitive feel in a number of commonly used operators, as well as a possible reduction in the overall number of operators required.

These improvements are achieved by avoiding the complicated and messy business of trying to *guess* the user's intentions about data from its chosen representation, and instead asking the programmer to express this information *explicitly*.

> "If you have two bits of information to represent, use two bits to represent it.
> Neither coincidence nor convenience justifies overloading a single bit."
> — Pitman's Two-Bit Rule

Acknowledgments

Innumerable legions of my co-workers at Harlequin rushed to help me search out the EBCDIC character code for capital A; I shall be forever indebted to these fine and dedicated individuals. John Aspinall and Scott McKay provided advice about the CLIM examples. Chris Stacy and Rebecca Spainhower read early drafts of this document and provided useful feedback.

References

[CL84] Guy L. Steele Jr., *Common Lisp: The Language*, Digital Press (Burlington, MA), 1984.

[CL93] Kent Pitman and Kathy Chapman (editors), *draft proposed American National Standard for Information Systems—Programming Language—Common Lisp*, X3J13 document 93-102, 1993. Available by anonymous FTP from `"/pub/cl/dpANS2"` on `PARCFTP.Xerox.COM`.

[CLIM92] Scott McKay and Bill York, *Common Lisp Interface Manager Release 2.0 Specification*. Available by anonymous FTP from `"/pub/clim"` on `Cambridge.Apple.COM`.