Manipulating Sets in Common Lisp*

Christop[illegible] R. Eliot
Computer and Information Science Department
University of Massachusetts, Amherst, MA 01003

May 4, 1989

Abstract

Sets are fundamental theoretical elements and yet Common Lisp does not provide any comprehensive representation for them. Several representations for sets are available, but they have serious problems. This paper defines a small family of primitive operations and uses them to evaluate existing Common Lisp set representations. This analysis characterizes how well sets are supported and suggests how the current situation could be improved.

1 Introduction

Sets are a basic mathematical and algorithmic concept. Several Common Lisp data types can be used to represent sets However, all of these representations have serous drawbacks that could be eliminated.

Sets can arise in many practical applications which dictate different representations. For example, a *MAKE* facility is likely to manipulate sets of files represented as sorted lists of pathnames. An assumption based truth maintenance system (ATMS) [2] is implemented using sets of assumptions represented with bitvectors. An expert system may represent a set of rules using an unsorted list of the names of the rules.

*This work is supported by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, the Air Force Office of Scientific Research, Bolling Air Force Base, District of Columbia 20332, under contract F30602-85-C-0008, and by a contract with Ing. C. Olivetti & C.

These examples illustrate several points. Sets typically have considerable structure in addition to their set theoretic properties. This structure may be as important as the set structure. Hence we believe that a separate data type for sets will be less useful than properly supporting the existing representations.

Applications vary in the degree to which data are manipulated as sets. At one extreme the sets are left implicit as the set elements are the only interesting datum. Structures or frames or objects are often manipulated on a purely local basis without ever explicitly constructing the set of objects. At the other extreme some systems (such as an ATMS) are primarily concerned with the sets per se, and care very little about the identity of the elements. In between are programs that examine the internal structure of set elements, and also require simple set manipulation operations, such as mapping and searching.

These three usage patterns correspond to three typical representations. Structures are used when the elements are primary. The set is represented implicitly by the pattern of links between structures, but cannot be manipulated directly. *Bitvectors* are often appropriate when the set structure is primary. When usable this representation makes set operations very fast, but accessing additional information about set elements is more difficult. Intermediate usage pattern generally are implemented using lists as a set representation. However, close examination of the latter two representations reveals some serious difficulties.

2 Representing Sets in Common Lisp

There are many ways to represent sets in Common Lisp each having different computational properties. Some representations can manipulate arbitrary data structures as set members. Other representations tradeoff simplicity and transparency for computational advantages. The choice depends upon the algorithm and expected use of the program.

Lists are ideal representations for many purposes because they require no auxiliary data structures nor extra preparations. When set operations are used intensively it may be worthwhile to consider more efficient representations. A finite universe of elements can be efficiently represented using a bitvector. Set operations can be several orders of magnitude faster when bitvector representations are used since the representation allows sets to be manipulated in chunks as large as the arithmetic registers.

Some finer distinctions can be made among these two broad categories of representation. Lists are often canonicalized by removing duplicate elements. If the list is also sorted then set equality corresponds to the Common Lisp EQUAL predicate and unions become simple merge operations[1].

Abstract bitvectors correspond with two Common Lisp data types. Integers can be manipulated as bitvectors using the logical operations on numbers. A one dimensional array of bits can also be used[2]. Integers have some disadvantages as a set representation. In most applications small integers are far more common than large ones and most implementations are optimized accordingly. However, in representing sets this is not true and so non-optimal performance can be expected. More seriously, there is no control over memory allocation for numbers so there is no way to prevent allocation of memory for intermediate results.

In contrast, bitarrays do not have these problems. Arrays can be modified in place, if desired, and so a carefully written program need not allocate memory for intermediate results. Bitarrays have their own problems, however. The most fundamental problem arises from the restriction of bitarray functions to arrays of matching rank *and dimenensions* [6, P.294]. Consequently adding an element to the universe requires extending every previously created set. In effect, bitarrays can only represent sets when the size of the universe of elements is bounded ahead of time.

3 Functionality

Representational ability may be evaluated systematically using an abstract family of primitive operations. Each representation is measured in terms of how many primitive operations are supported and how efficiently. Figures 1 and 2 list definitions of the operations and predicates used for this analysis. Seven operators and seven predicates comprise the family of primitive operations. These operations are commonly used in set theory [7], mathematics, computation theory [1], [5] and practical programming.

The cartesian-product may not seem to be a primitive operation. However, it arises naturally in Lisp applications. Suppose three variables, a, b

[1] This correspondence between EQUAL and Set Equality depends upon assumptions about the interpretation of Equality between set elements. However, when these assumptions are violated Set Equality can still be implemented using a construct involving the Every sequence function.

[2] We will use the term "bitarray" to distinguish these from abstract bitvectors.

Union(A, B)	$\equiv \{x \mid x \in A$ or $x \in B\}$
Intersection(A, B)	$\equiv \{x \mid x \in A$ and $x \in B\}$
Difference(A, B)	$\equiv \{x \mid x \in A$ and $x \notin B\}$
Add(y, S)	$\equiv \{y\} \cup S$
Delete(y, S)	$\equiv \{x \mid x \in S$ and $x \neq y\}$
Cardinality(S)	$\equiv$ number of elements in S
Cartesian-Product(A, B)	$\equiv \{\langle x, y\rangle \mid x \in A, y \in B\}$

Figure 1: The Primitive Operations.

Member(x, S)	$\equiv x \in S$
Subset(A, B)	$\equiv A \subseteq B$
Proper-Subset(A, B)	$\equiv A \subset B$
Intersects(A, B)	$\equiv A \cap B \neq \emptyset$
Disjoint(A, B)	$\equiv A \cap B = \emptyset$
Equal(A, B)	$\equiv A = B$
Empty(S)	$\equiv S = \emptyset$

Figure 2: The primitive predicates.

and c have values known to be in the sets A, B, and C respectively. Then the values of the triple, $\langle a, b, c\rangle \in A \times B \times C$. Any system which searches for consistent sets of bindings is likely to use this operation at some point. Embedded rule based systems, logic programming languages and planners all search for consistent sets of bindings. The QSIM system [4] uses the cartesian product in the simulation algorithm to combine the possible transitions for individual *quantities* into consistent global predictions of the next qualitatively distinct state. The ATMS label propagation algorithm [2] conceptually uses a cartesian product, but due to the highly optimized data structures the operation cannot be isolated in the code. This suggests that some form of generalized cartesian product operator might be possible.

Figure 3 shows which of these primitive operations correspond to Common Lisp functions. No representation supports all operations, even ommiting the cartesian product and proper subset. Surprisingly, there are more set primitives defined for numbers than for lists. In general the set operations are well supported, but the predicates are not.

Some of the missing functions can be implemented on top of Common Lisp. For example, the superset is just subset with the arguments reversed. However, some operations require either a complex implementation or waste-

Operation	CList	SCList	NCList	BA	NUM
union	+	+	+	+	+
intersection	+	-	+	+	+
difference	+	-	+	+	+
add	+	-	+	+	+
delete	+	+	+	+	+
size	+	+	-	-	+
member	+	+	+	+	+
subset	+	+	+	-	-
proper-subset	-	-	-	-	-
intersects	-	-	-	-	+
equal	-	+	-	+	+
null	+	+	+	-	+

CList	Unsorted Canonical List
SCList	Sorted Canonical List.
NCList	Non Canonical List.
BA	Bitarrays
INT	Integers, used as bitvectors.

Figure 3: Common Lisp set primitives

fully constructing intermediate results. For example, performing a subset test using integer set representations requires construction of the set difference, which may be a bignum, only to test whether it is empty. Finally, some opertions are essentially impossible to implement portably. For example, determining that a bitarray represents the null set requires inspecting every bit. A portable implementation must test each bit separately, but an implementation specific implementation can test groups of bits as large as the word size of the machine. Hence an implementation specific implementation can be orders of magnitude faster.

4 Observation

Common Lisp should be extended to provide better representational support for sets. Both list and bitvector representations are desirable since they have different computational properties. The restriction of Bitarray operations to vectors of equal length seriously diminishes the utility of these operations for set representation and so should be removed. Finally facilities for converting sets among the representations should be available. These additions would dramatically improve Common Lisp's set representation capabilities. Commercial systems that require high performance would be better supported by bitvector representations. Academic users could more easilly show the correspondence between their theories and implementations. And all users could more easilly exploit published theoretical ·scriptions of algorithms.

5 Proposal for Common Lisp

Factors quite removed from theoretical considerations must be considered in a language specification. Compatibility and simplicity are very important. Compatibility involves both functional compatibility with existing programs, and philosophical consistency within the language. This proposal attempts to strike a proper balance in this regard. It is separated into a core section containing a minimal set of functions required to obtain full support of sets as defined earlier and a cosmetic section mostly containing alternate names to make code more readable. All of the new functions have been implemented in Common Lisp[3].

5.1 Core Proposal

Ten new functions, and some modifications of existing functions form the core proposal. The only functional incompatibilities introduced by these changes are naming conflicts.

5.1.1 List Representations

The changes needed to support list representations are addition of set predicates for intersection and equality, and extensions to the intersection, difference and adjoin operators to handle sorted canonical lists. Specifically these functions are extended to handle a new keyword argument :sorted-by, which specifies a sorting predicate. This argument deaults to NIL and must be either NIL or a valid second argument to the SORT function. When a non-NIL :sorted-by argument is supplied these functions guarantee that the resulting list will be sorted in the implied order, provided the list argument(s) were initially properly sorted. An actual implementation may simply perform the

[3]Not necessarilly very efficiently however.

basic operation and sort the result, or it may perform an appropriate merge operation.

intersectsp (list1 list2 &key :test :test-not :key)

set-equal (list1 list2 &key :test :test-not :key)

These two predicates operate uniformly on all list representations of sets, and implement the corresponding set predicate. The arguments are defined as for subsetp.

5.1.2 Bitvector Representations

Supporting bitvector representations requires adding most of the set predicates. Furthermore, the bitarray operations must be extended to handle vectors of differing lengths. The rule of thumb that applies is that the smaller array must be treated as if it were extended with zeros to be as big as the larger array. Set representations only require one dimensional arrays and so strictly this extension does not have to apply to higher-dimensial cases. However, it seems that this extension would be useful in two dimensional arrays for graphics operations. Color graphics might be represented using three dimensional arrays. For example, a glyph in some character set might be represented as a small bitarray copied into a picture with bit-ior or bit-xor. The current Common Lisp specification [6, P.294] makes this unworkable.

The additional functions required to support bitvector representations are:

bit-subsetp (bit-array1 bit-array2)
logsubsetp (N M)
bit-intersectsp (bit-array1 bit-array2)
bit-set-equal (bit-array1 bit-array2)
bit-zerop (bit-array)
bit-count (bit-array)

Bit-set-equal is just like equalp except that it treats vectors as infinitely extended with zeros for comparison purposes.

5.2 Conversions

In many situations it is desirable to convert sets from one representation to another. For example, a program may internally manipulate sets using bitvectors for efficiency, while using a list representation for input and output. Some of these conversions already are supported by Common Lisp. For example, a non-canonical list can be converted into a canonical list using the remove-duplicates function. Once the set is represented as a canonical list, it may be converted into sorted canonical form using the sort function. Conversion in the other direction is not needed since a sorted canonical list can be used wherever a non-canonical list representation is acceptable.

However, conversions to and from the bitvector representations are not supported. Common Lisp provides no facility to convert a set represented using a bitarray into a set represented as an integer, even though some implementations use bitarrays internally to represent large integers [3] making implementation of this primitive trivial. Four new functions suffice to allow sets in any representation to be converted into equivalent sets in any other representation.

set-vector-to-integer (vect)
set-integer-to-vector (n)
set-vector-to-list (vect map)
set-list-to-vector (list map &key test key)

The last two functions require a map argument, which is a general vector in which the list representations of set elements are stored in positions corresponding to the placement of bits in the bitarray representations. For example:

(Set-list-to-vector '(C A) #(A B C)) => #*101

5.3 Cosmetic Additions

A number of set manipulation functions are trivial variants of those proposed already. However, for consistency among programs it is better to provide these as part of the language. For this reason it is desirable to provide superset and proper-subset functions in addition to the minimally required subset functions.

supersetp (list1 list2 &key key test test-not)
logsupersetp (n m)
bit-supersetp (bit-array1 bit-array2)

proper-subsetp (list1 list2 &key key test test-not)
logproper-subsetp (n m)
bit-proper-subsetp (bit-array1 bit-array2)

proper-supersetp (list1 list2 &key key test test-not)
logproper-supersetp (n m)
bit-proper-supersetp (bit-array1 bit-array2)

The proper-subset functions can be formed from the subset and equal functions. However, it is possible to implement proper subset using a single scan over the data, while the composite form implies two scans. Therefore adding the primitives for the proper subset operations might result in some efficiency gains. The proper superset functions are trivial variants of these.

Furthermore, to maintain the illusion that sets are being manipulated rather than arrays or integers the basic union, intersection and difference functions should be given synonyms that reflect their usage. The following table defines synonyms for six bit and integer functions. The synonomous names provide better mnemonic reference to their function as set manipulation functions.

root	name	synonym
log	ior	union
bit-	and	intersection
	andc2	difference

These cosmetic changes require the addition of twelve more symbols, but no significant implementation effort.

References

[1] Aho, A. V., Hopcroft, J. E. & Ulman, J. D., "The design and analysis of Computer Algorithms," Addison-Wesley, Reading Mass. (1974).

[2] de Kleer, J., "An Assumption Based Truth Maintenance System," Artificial Intelligence, Elsevier Publishers, North-Holland (1986).

[3] Burke, G. S., Carrette, G. J. and Eliot, C. R., "NIL reference manual," MIT/LCS/TR-311 (1984).

[4] Kuipers, B.J., "Qualitative Simulation," Artificial Intelligence Vol. 29 No. 3, pp 289-338 (1986).

[5] Lewis, H. R., & Papadimitriou, C. H., "Elements of the Theor[illegible] Computation," Prentiss-Hall, Inc., Englewood Cliffs, New Jersey (1981).

[6] Steele, G. L., "Common Lisp: the language," Digital Press (1984).

[7] Suppes, P., "Axiomatic Set Theory," Dover publications, New York, (1972).

Solutions

Let us note *cc* the value of `call/cc`. Continuations will be noted *k* as usual. Various indices will clearly separate history of values.

First Puzzle

Let us trace the computation of k_0(`call/cc`$_1$ `call/cc`$_2$) where k_0 is the embedding continuation. The evaluation of the terms lead to $k_0(cc_1\ cc_2)$ that is to say, by virtue of *cc*, to $k_0(cc_2\ k_0)$ and, by another virtue of *cc*, $k_0(k_0\ k_0)$ and eventually k_0.

Therefore `(call/cc call/cc)` can be nicknamed `(the-continuation)` since it returns the current continuation. More precisely as shall be seen in the next puzzle, it feeds the current continuation with the current continuation.

Had `call/cc` be "jumpy", the result would have been the "black hole" continuation. Let us note $k_\perp$ this very continuation, something like $\lambda\epsilon\cdot\perp$ which swallows whatever value sent to it but also freezes all computations. The computation of k_0(`call/cc`$_1$ `call/cc`$_2$) now leads to $k_0(cc_1\ cc_2)$ and then to $k_{\perp_1}(cc_2\ k_0)$ i.e. $k_{\perp_2}(k_0\ k_{\perp_1})$ and eventually $k_{\perp_1}$, the black hole continuation. By the way it is unclear that (eq? $k_{\perp_1}$ $k_{\perp_2}$) is true, since nothing prevents black hole continuations to be the same all the time !

CHRISTIAN QUEINNEC NITSAN SÉNIAK