Lisp and Symbolic Computation **5**,3 (Sept. 1992), 157-190.

A Decision Procedure for Common Lisp's `SUBTYPEP` Predicate

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, California 91436
(818) 986-1436 (818) 986-1360 (FAX)

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663.

Abstract

Common Lisp [CL84,CL90] includes a dynamic datatype system of moderate complexity, as well as predicates for checking the types of language objects. Additionally, an interesting predicate of two "type specifiers"—`SUBTYPEP`—is included in the language. This `subtypep` predicate provides a mechanism with which to query the Common Lisp type system regarding containment relations among the various built-in and user-defined types. While `subtypep` is rarely needed by an applications programmer, the efficiency of a Common Lisp implementation can depend critically upon the quality of its `subtypep` predicate: the run-time system typically calls upon `subtypep` to decide what sort of representations to use when making arrays; the compiler calls upon `subtypep` to interpret user *declarations*, on which efficient data representation and code generation decisions are based.

As might be expected due to the complexity of the Common Lisp type system, there may be type containment questions which cannot be decided. In these cases `subtypep` is expected to return "can't determine", in order to avoid giving an incorrect answer. Unfortunately, most Common Lisp implementations have abused this license by answering "can't determine" in all but the most trivial cases. *In particular, most Common Lisp implementations of* `SUBTYPEP` *fail on the basic axioms of the Common Lisp type system itself* [CL84,p.33]. This situation is particularly embarrassing for Lisp—the premier "symbol processing language"—in which the implementation of complex symbolic logical operations should be relatively easy. Since `subtypep` was presumably included in Common Lisp to answer the *hard* cases of type containment, this "lazy evaluation" limits the usefulness of an important language feature.

This paper shows how those type containment relations of Common Lisp which can be decided at all, can be decided simply and quickly by a decision procedure which can dramatically reduce the number of occurrences of the "can't determine" answer from `subtypep`. This decision procedure does *not* require the conversion of a type specifier expression to conjunctive or disjunctive normal form, and therefore does not incur the exponential explosion in space and time that such a conversion would entail.

The lattice mechanism described here for deciding `subtypep` is also ideal for performing *type inference* [Baker90]; the particular implementation developed here, however, is specific to the type system of Common Lisp [Beer88].

Categories and Subject Descriptors: Lisp, dynamic typing, compiler optimization, type inference, decision procedure.

1. Introduction

The Common Lisp functions `typep` and `subtypep` are the basic predicates of the Common Lisp type system [CL84,p.72]. Loosely speaking, `(typep x y)` poses the question of whether the *object* `x` is an *element* of the *type* `y`, while `(subtypep y z)` poses the question of whether the *type* `y` is a *subset* of the *type* `z`. While `typep` is heavily used to distinguish object types at run-time, and indeed the common `typecase` construct [CL84,p.118] depends upon `typep`, `subtypep` is considered by most Common Lisp applications programmers to be an obscure language function of only theoretical interest. However, if these programmers were aware that the quality of the optimization of their programs depended critically on the quality of the implementation of `subtypep`, they might take more notice. For example, these programmers might become exasperated if a program with a variable declared as a `bit` (taking on only the integer values 0 or 1) actually runs *slower* than when the same variable is declared as a `fixnum` (taking on any small integer values). They would be similarly distressed by an implementation which gave them a general vector of type `(array t (*))`, when `make-array` was called with an `element-type` of `(integer 0 63)`, even though the Lisp system was capable of providing specialized arrays of signed 8-bit integers.[1] These problems can both be traced to deficiencies in the particular Lisp system's implementation of `subtypep`.

The problem that a Common Lisp implementation is typically trying to solve when calling `subtypep` is to determine the storage requirement for a given object, as specified by a Common Lisp type specifier expression. The more accurate `subtypep` is, the smaller the storage required for the user's program, and the faster it can be as a result of manipulating more compact objects.

Optimizing Common Lisp compilers also attempt to remove type checks from a user's program, whether they were inserted by the programmer, the expansion of a macro, or a code expansion during code generation. Eliminating such type checking is essential to achieving high execution speed on modern RISC architectures. If the compiler can prove, by calling `subtypep`, that the type check will always be satisfied, then the type check is redundant code, and can be eliminated without any ill effects. Once again, the quality of the implementation is directly related to the quality of `subtypep`.

Many programmers from outside the Lisp community are accustomed to including type declarations in their programs, and start programming in Lisp by including complete type declarations. This information can be valuable for the compiler, since compilers for non-Lisp architectures can often make use of declarations to improve execution speed. Supplying the most specific declarations can also be valuable for program maintenance, because such declarations provide excellent documentation. However, in many Common Lisp implementations this attention to detail is penalized. Highly specialized declarations may actually make programs run *slower* than less specialized declarations, because poor implementations of `subtypep` may not be capable of correctly interpreting the more complex type specifier expressions.

Thus, a high-quality `subtypep` implementation is a prerequisite for a high-quality Common Lisp implementation, especially when the host processor is an architecture not well suited for dynamic typing. Yet the level of `subtypep` implementation among current commercial Common Lisp's is not high. Informal tests on various implementations showed problems ranging from ignorance—almost always returning "don't know"—to inconsistency—violating the transitivity of the `subtypep` relation. However, even the best ones fail to decide the basic set of axioms covering Common Lisp datatypes [CL84,p.33].

This paper gives a *decision procedure* for `subtypep`, which will answer any question posed to `subtypep` that does not include the `satisfies` clause. This restriction is forced by the fact that the general solution to `subtypep` with the `satisfies` clause is recursively undecidable.

We describe the datatypes of Common Lisp in section 2, describe the language of type specifier expressions in section 3, and discuss the possible meanings of Common Lisp datatypes in section 4. In section 5 and 6, we gain additional insight into the nature of the Common Lisp type system and `subtypep` by showing where an *ad hoc* approach to deciding `subtypep` fails, and we see why

---

[1]The new function `upgraded-array-element-type` is supposed to fix this problem, but only succeeds in hiding the problem at a lower level. Since this new function is required to be lattice-monotonic [CL90,67-68], and since the upgraded type of `'bit` is required to be `'bit` [CL90,p.33], we can use this function to test for the emptiness of an arbitrary type specifier—a problem we later show to be equivalent to `subtypep`.

```
(defun type-null (x)
  (values (and (eq 'bit (upgraded-array-element-type `(or bit ,x)))
               (not (typep 0 x))
               (not (typep 1 x)))
          t))
```

by considering other hard questions that could be decided by using `subtypep` as a subroutine. Section 7 deals with the `subtypep` decision procedure itself, while section 8 indicates its complexity. Finally, section 9 discusses the sensitivity of the decision procedure to the particulars of the Common Lisp, and section 10 concludes.

2. Common Lisp data types

Before we can decide the solution to the type containment problems of Common Lisp, we must first understand its type system. Common Lisp [CL90] has a moderately complex type system, including limited precision integers, unlimited precision integers, rational numbers, floating-point numbers in four different formats, complex numbers of various types and sizes, characters of considerable diversity, a host of specialized types such as `compiled-function`'s and `readtable`'s, arrays of various ranks, sizes and element types, and user-defined `structure`'s and `class`'es (*records* in other languages). Common Lisp prescribes certain containment and disjointedness relations among the built-in types for *all* implementations, and allows certain other containment and disjointedness relations to depend upon the implementation [CL90,p.37-41].

For example, Common Lisp prescribes that the types `real` (which contains both `rational` and `float`) and `complex` shall be pairwise disjoint subtypes of `number`. However, depending upon the implementation, there may be numeric elements which are *not* real or complex. In fact, an interesting question for a "portable" Common Lisp program to ask an implementation is whether there are any numbers other than `real` and `complex` numbers. This question can be posed to `subtypep` by asking whether `number` is a subset of the union of `real` and `complex`.

The Common Lisp type system has its nits. For example, while "`3/1`"—the rational number with numerator `3` and denominator `1`—is a member of `rational`, because `3/1` is an integer, and all integers are rational numbers, "`#C(3/1 0)`"—the complex number with real part `3/1` and imaginary part `0`—is not a member of `(complex rational)`, because `#C(3/1 0)` is normalized to the rational number `3/1`, and the Common Lisp standard says that the types `rational` and `complex` are disjoint, and therefore that `(complex rational)`—a subtype of `complex`—is disjoint from `rational`, as well.

Figure 1 is a chart classifying Common Lisp datatypes into 5 different "kingdoms".

flat elementary types	`symbol`, `character`, `readtable`, `complex`, etc.
numeric ranges	`(integer * *)`, `(ratio * *)`, `(short-float * *)`, etc.
array types	`(array` *rep-type1* `*)`, `(array` *rep-type2* `(* 3))`, etc.
`defstruct` type hierarchy	*structure-name-1*, *structure-name-2*, etc.
CLOS type partial order	*class-name-1*, *class-name-2*, etc.

Figure 1. Chart of Common Lisp data type kingdoms.

3. Common Lisp type specifier expressions

Type containment questions in Common Lisp are posed by calling `subtypep` with two "type specifiers", which are expressions in a formal language. These type specifier expressions denote the types whose containment is to be decided. The power of `subtypep` is directly related to the "expressiveness" of this type specifier language; on the other hand, the more expressive the language, the more difficult it is to decide the questions.

The Common Lisp type system allows for the construction of "type specifier" expressions which include the basic type symbols, as well as a number of operations involving those type symbols and (recursively) other type specifier expressions. Thus, type specifiers consist of basic type symbols such as `integer`, `float`, `character`, numeric ranges such as `(integer 3 6)`, meaning the integers {`3`,`4`,`5`,`6`}, and array types such as `(array float (3 *))`, meaning all 2-dimensional arrays whose containers are specialized to hold floating point numbers, and whose first dimension must be `3`. Furthermore, a type can be specified by the `member` construct, where `(member` *a b c*`)` is just the finite set of Lisp objects {*a*,*b*,*c*}. A type can also be specified operationally using a predicate, as in `(satisfies oddp)`, meaning all Lisp objects *x* such that `(oddp` *x*`)` is true.

Finally, simple type specifiers can be composed into compound type specifiers by means of the connectives `and`, `or`, and `not`.

The Common Lisp type system allows the basic type system to be extended by the user in three ways through the use of `deftype`, `defstruct`, and `defclass`. The first, `deftype`, is simply a macro facility for type specifier expressions; thus, no new functionality is introduced through `deftype`, but brevity is facilitated. The other two ways of type system extension, `defstruct` and CLOS's `defclass`, allow the creation of new types of Lisp objects which are disjoint from every other type (except when otherwise specified). As we will later show, the change in functionality of the type system introduced by `defstruct` types is minimal and relatively self-contained.

Common Lisp has recently been extended with the "Common Lisp Object System", or CLOS, for short [Bobrow88]. CLOS objects (*instances*) are elements of *classes*, which are different from Common Lisp *types*, but which map into the Common Lisp type system. Loosely speaking, this mapping from classes to types strips all *intensional* meaning from the class, and produces the *extensional* set of elements (called *instances*) of the class. Given this interpretation, we will see that CLOS usually causes no problems for the `subtypep` decision procedure, as we can continue to work with only the *type* structure and not the *class* structure of CLOS objects.

Common Lisp also defines two additional type specifier constructs: `function` and `values`. However, these constructs are expressly prohibited from calls to `typep`, and hence from calls to `subtypep` [CL90,p.57]. If these specifiers were not prohibited by Common Lisp from appearing in calls to `subtypep`, they would make a decision procedure for `subtypep` impossible. For example, `subtypep` could then have been asked about the functional equivalence of two lambda-expressions, a problem which is known to be undecidable.

Type Names	`integer`, `number`, `symbol`, `null`, `t`, etc.
Numeric Ranges	`(integer` *low high*`)`, `(float` *low high*`)`, etc.
Arrays	`(array bit (3 *))`, `(string` *size*`)`, etc.
Individuals	`(member` *object1 object2 ...*`)`
Predicates	`(satisfies` *predicate-name*`)`
Connectives	`(not` *specifier*`)`, `(and` *specifier1 specifier2*`)`, `(or` *specifier1 specifier2*`)`

Figure 2. Chart of Common Lisp "type specifier" expressions.

4. The meaning of a Common Lisp "type"

So far, we have a notion of a datatype and a language for specifying datatypes, but have not yet related the two. In this section, we give meaning to type specifier expressions by modelling the syntactic constructs of the language with operations on datatypes.

There are several ways to interpret the notion of *type*. One can interpret *type* as a *set* of elements, in which it has the *extensional* meaning of a set—an *extensional type* is no more or less than an unordered list of its elements. One can also interpret a *type* as a member of a mathematical *lattice* (see Appendix I) in which the subtype relation is a purely formal concept, and in which the type names themselves have some meaning; i.e., this is an *intensional* notion of type [Scott76]. The types-as-lattices concept has been utilized for years as a representation of the *meaning* of a program, and by compilers for performing data flow analysis (including *type inference* [Kaplan80]). The Common Lisp standard says in no uncertain terms that "a data type is a (possibly infinite) *set* of Lisp objects" ([CL90,p.12], italics added for emphasis). Thus, in determining the meaning of `subtypep`, a Common Lisp implementor is required to implement the *extensional*, or *set*, notion of type, and hence answer `subtypep` questions from an extensional point of view.

How do these notions of intensional and extensional types differ in practice? One way the notions differ is in whether `subtypep` questions are to be answered for *this* implementation, or for *all* Common Lisps. Thus, one Common Lisp may implement a numeric type which is not `real` and not `complex` (e.g., exact representations of irrational extensions implemented through symbolic algebraic techniques [Loos83]), hence in this implementation, `number` is not intensionally or extensionally the same as the union of `real` and `complex`. However, if no such numeric elements exist in another implementation of Common Lisp, then in that implementation, `number` *is* extensionally the same as the union of `real` and `complex`, even though `number` is *not* intensionally the same. Common Lisp clearly intends this extensional view of *types*.[2]

Thus, the extensional notion of types is equivalent to the notion of containment of sets, so the familiar laws of *Boolean algebra* hold (see Appendix I). Since Boolean algebras are also lattices, the lattice "≤" operation becomes equivalent to *subset*, which is therefore the model for `subtypep`. Additionally, the lattice operations of *least upper bound* and *greatest lower bound* become equivalent to the *union* and *intersection* of sets, respectively, and these can also be utilized wherever lattice operations are required (e.g., for type inference [Kaplan80], [Beer88]).

Several additional complications regarding `subtypep` are more easily dealt with using the Boolean algebra interpretation of types, due to its isomorphism with the standard Boolean algebra of *sets*: namely, the ability to ask questions about particular elements (utilizing the `member` construct), and the ability to talk about the complement of a set (using the `not` construct). Such questions would have to be dealt with on an *ad hoc* basis with a lattice interpretation, or not at all.

The algebraic laws for the manipulation of Boolean algebras are simpler and more familiar than those of lattices (see Appendix I). The intuition most people have of lattice manipulations is very restricted, and one must be very careful not to use Boolean thought processes when they are not appropriate. For example, most people would assume that it is appropriate to *distribute* ("multiply out") `and` over `or` in the type specifier expression `(and float (or single-float null))` to produce the result: `single-float`. In the obvious data-type lattice, however, the answer without the distribution would have been `float`. This curious result obtains because the obvious Common Lisp type lattice is *non-modular* and hence not *distributive;* therefore it cannot be Boolean, since Boolean lattices are distributive [MacLane67].

5. Current implementations of `subtypep`

The Common Lisp standards committee noted that "[`subtypep` questions involving] `and`, `or`, `not`, and `member` are ... very messy to deal with" [CL90,p.97-98]. We agree that the Common Lisp type system itself is very messy, but this messiness can be dealt with through a uniform bit-vector representation in which the messiness is confined to the contents of these bit-vectors. However, in order to understand the need for this representation, it is instructive to examine the *ad hoc* approach typical of current implementations.

---

[2]There could also conceivably be an alternative *modal* view of Common Lisp types, where the modal constructs "may" and "must" would represent the differences in implementations (or *worlds*, see [Hughes68]). "May" would be interpreted as "there could conceivably exist an implementation", and "must" would be interpreted as "for all conceivable implementations". Under this modal view, there would be *undecidable* `subtypep` questions even in the absence of `satisfies`, because there would be specifiers *t1* and *t2* such that the answer to the question `(subtypep` *t1 t2*`)` would differ depending upon the implementation, and hence it would be impossible for `subtypep` to answer with certainty any question for which

```
(or (must (subtypep t1 t2)) (must (not (subtypep t1 t2))))
```

was not true. We reject this modal view as not being useful for a program to want to query because most programs would be more interested in *this* implementation—i.e., the one the program is running on, or being compiled for.

```
(defun o-t-subtypep (x y)
  ;  o-t-subtypep  =  "obviously-trivial-subtypep"
  (cond
    ((and (eq x 'fixnum) (eq y 'float))  ;  Do all pairs of type symbols.
     nil)
    ((and (eq x …) (eq y …))            ;  ad nauseum.
     t/nil)
    ((eq x nil) t)                       ;  1st arg empty disjunction.
    ((eq (car x) 'or)                    ;  1st arg disjunction easy.
     (every #'(lambda (nx)
                (o-t-subtypep nx y))
            (cdr x)))
    ((eq (car x) 'member)                ;  ditto.
     (every #'(lambda (xelt)
                (typep xelt y))
            (cdr x)))
    ((eq y t) t)                         ;  2nd arg empty conjunction.
    ((eq (car y) 'and)                   ;  2nd arg conjunction easy.
     (every #'(lambda (ny)
                (o-t-subtypep x ny))
            (cdr y)))
    ((and (eq (car x) 'not)              ;  Contrapositive easy.
          (eq (car y) 'not))
     (o-t-subtypep (cadr y) (cadr x))
    ((eq x t) ???)                       ;  1st arg empty conjunction.
    ((and (eq (car x) 'and)              ;  Partial for 1st arg conjunct.
          (some #'(lambda (nx)
                    (o-t-subtypep nx y))
                (cdr x)))
     t)
    ((eq y nil) ???)                     ;  2nd arg empty disjunction.
    ((and (eq (car y) 'or)               ;  Partial for 2nd arg disjunct.
          (some #'(lambda (ny)
                    (o-t-subtypep x ny))
                (cdr y)))
     t)
    ((eq (car y) 'member) ???)
    (...)                                ;  ???
```

In this example, we see that the hacker approach fails to make progress after the *monotonicity* rule has been used in every conceivable way. In order to make any further progress, we must make use of the other properties of Boolean algebras, and this will require breaking up the problem along different dimensions.

The incompleteness of the *ad hoc* approach is not its only problem. Its soundness is also difficult to demonstrate. With a program of this complexity (some implementations go on like this for 4-5 pages), it is difficult to determine its correctness, and indeed some Common Lisp `subtypep`'s violate the basic axioms of a *partial order*—like transitivity. Yet even after all of this work, most implementations still cannot decide the basic propositions that the Common Lisp specifications pose as axioms.

6. What is the power of `subtypep`?

In this section, we explain the difficulties in properly implementing `subtypep`, by showing what other questions that can be decided using `subtypep` as a subroutine. This "reduction" technique thus proves that deciding `subtypep` is at least as hard as deciding any of these other questions. <u>These reductions, however, depend critically on the "set" interpretation for "type".</u>

Using `subtypep`, one can ask whether:
- a type is empty (`nil`)
- two types are disjoint
- two types are equal
- an object is an element of a given type
- two objects are identical (`eql`)
- an object is a member of a given finite set
- one finite set is a subset of another finite set
- a number is positive
- one number is less than or equal to another.

The code below shows each of these reductions of the given question to a `subtypep` question.

```
(defun type-null (x) (subtypep x nil))

(defun type-disjoint (x y) (subtypep `(and ,x ,y) nil))

(defun type-equal (x y)
  (multiple-value-bind (x<=y okxy) (subtypep x y)
    (multiple-value-bind (y<=x okyx) (subtypep y x)
      (values (and x<=y y<=x) (and okxy okyx)))))

(defun typep (x y) (values (subtypep `(member ,x) y)))

(defun eql (x y) (values (type-equal `(member ,x) `(member ,y))))

(defun member (x y) (values (subtypep `(member ,x) `(member ,@y))))

(defun subsetp (x y) (values (subtypep `(member ,@x) `(member ,@y))))

;;; The following are restricted to rational numbers only.
(defun plusp (x) (values (subtypep `(rational ,x *) '(rational (0) *))))

(defun <= (x y) (values (subtypep `(rational * ,x) `(rational * ,y))))
```

One can even perform "pattern-matched" database retrieval by storing the database records as a large union of arrays of equal rank whose declared dimensions encode the data; `subtypep` can determine if a given pattern-encoded-as-array-type matches any of the arrays in the union. The code below implements this form of database retrieval using `subtypep`. It works by storing the data records as elements of a relation. A query consisting of a record without "`*`" components is decided by determining whether the element sought is *not* in the *complement* of the union of stored records; if so, then it *is* in the database. A query consisting of a record with "`*`" components defines another relation on the same domain as that of the database. If this query relation is not completely contained within the complement of the stored relation, then there exists an element of the query that is *not* in the *complement* of the union of the database records, hence *is* in the union, and hence *is* in one of the records.

```
(defvar *database* nil)

(defun store-record (record)
  ; record = (<i> <j> ... <n>), i,j,k non-negative integers.
  (push `(array t ,record) *database*))

(defun retrieve-record (record)
  ; record = (<i/*> <j/*> ... <n/*>), i,j,k non-negative integers
  (not (subtypep `(array t ,record)
                 `(not (or ,@*database*)))))
```


However, there are certain questions one *cannot* ask with `subtypep`. `subtypep` cannot be asked what *size* a set is (other than whether it is empty); it cannot be asked to do arithmetic, it cannot be asked hypothetical questions. As a result, a decision procedure for `subtypep` does not have to deal with alternation of quantifiers, Skolem functions, or any other device that could cause multi-exponential inefficiency in a decision procedure.

Before going into the details of the implementation of `subtypep`, we notice that `subtypep` could have been defined in terms of the simpler type specifier predicate `type-null` (given below), if the language designers had been more confident of an efficient implementation. (*NB:* `type-null` is not the same as `null`, since the argument for `type-null` is evaluated as a type-specifier, while the argument for `null` is evaluated as a Lisp expression.)

```
(defun subtypep (x y)
  ; TYPE-NULL determines if its argument is
  ;  a type specifier for the empty type.
  (type-null `(and ,x (not ,y))))
```

7. The `subtypep` decision procedure

Our decision procedure for `subtypep` will divide and conquer. We will utilize separate decisions procedures for the *disjoint* "kingdoms" of data types in the Common Lisp type system.[3] By utilizing the fact that the Boolean algebra for the Common Lisp type system is the *Cartesian product* of the Boolean algebras for each of these disjoint "kingdoms" of types, we will then be able to handle the general case. (This approach is analogous to the use of the Chinese Remainder Theorem to solve problems involving integers by mapping the problems into several homomorphic images.)

In many cases, `subtypep` will depend upon `typep` for its answer. `typep` itself may be extremely complex in the general case—especially when CLOS types are involved. Nevertheless, `typep` is required by Common Lisp to be a decision procedure, so we can depend upon `typep`'s answers.

7.1 Implementing `subtypep` for primitive elementary types

Once we have focussed on the *set* interpretation of data type, it should be possible to answer subtype questions by simply comparing the membership of the sets. However, for the Common Lisp datatypes with large or infinite sets (e.g., `integer`), this implementation is either impractical or impossible. The work of the comparison process can be reduced by noting that due to the limitations of the language of type specifications involved, most elements of the various sets are *indistinguishable*, and that any member of such a group of indistinguishable elements will always be treated the same under all the set operations. These sets of indistinguishable elements are called *equivalence classes*. With the exception of ranges, arrays and user-defined types, the number of equivalence classes for Common Lisp types is *finite*. Thus, for the built-in types, excluding ranges, arrays and user-defined types, we can accurately answer any `subtypep` question with a simple subset test on two finite sets. With this reduction of the comparison problem, implementing `subtypep` involves the selection of a *representative* element from each of the equivalence classes of elements, and performing Boolean operations on these sets of representatives instead of on the whole sets.

The next step is to choose correct representative elements from the equivalence classes. We can choose representative elements using the same principle that medical students use to organize the wide variety of diseases and symptoms: *differential diagnosis.* We need each element to somehow be different from each other element, in the sense that some set (type) includes one but not the

---

[3]In some implementations, `defstruct` and `defclass` types may overlap with the elementary Common Lisp types—e.g., `stream`. Since we utilize the same mechanism for these three kingdoms, we can collapse these three kingdoms into one for the implementation of `subtypep`. However, we insist that numeric ranges and arrays be disjoint from each other and from the previous three kingdoms.

other. If we choose extra representatives from an equivalence class, they will slow down our decision procedure slightly, but will not cause any incorrect answers. However, if we fail to choose an element which distinguishes between two sets, then those sets may be incorrectly determined by `subtypep` to be equivalent, when in fact they are not. This problem will be examined later in more detail, under the `member` construct and user-defined type discussions.

In the case of the built-in Common Lisp types, the choice of representatives is straightforward. We must select at least one representative member of each built-in type (assuming that they have any members at all); a `fixnum`, a `bignum`, a `ratio`, a `short-float`, a `single-float`, a `character`, etc. We may need some additional elements when there are complex relationships; e.g., in the characters, where we need a `base-character`, a non-base `character`, and an `extended-character` which is not a `character`.

How many representatives are required? For the built-in types, not including ranges and arrays, fewer than 30 elements are required. Applying the standard trick of representing finite sets with integers interpreted as bit-vectors, then mapping union, intersection and complement onto `logior`, `logand`, and `lognot`, we can represent these sets using relatively small integers which fit into a single machine word—i.e., `fixnum`'s. Thus, not only can we answer `subtypep` questions involving the basic Common Lisp types accurately, we can also answer them quickly using a few logical operations on word-size quantities.

An example of an implementation of `subtypep` for built-in datatypes using this technique can be found in Appendix II.

7.2 How to handle the MEMBER construct

The `member` construct adds complexity to our problem by rendering the *a priori* selection of representatives, as described above, inadequate. Instead of asking questions about anonymous `short-float` or `base-character` elements, one can now ask questions about specifically *mentioned* elements. Using the `member` construct, one can also ask questions about anonymous arbitrary finite sets of Lisp objects, with no reference to the rest of the Common Lisp type system at all. Both of these forms of question can be seen in our emulation of `typep` and `subsetp` in section 5.

The trick to efficiently handling `member` is to *include the mentioned elements* as additional representatives for our finite sets. This requires two passes through the type specifier expression: the first to find all elements mentioned in `member` constructs and make them additional representatives; and the second to evaluate the type specifier expression using Boolean operations on bit-vectors. The only complication involves small sets (such as `null`, which has only one element—`nil`), and obvious elements such as the `fixnum` 0 (zero). To handle finite subset questions in which the *only* clauses are `member` constructs, we have to *register* our representatives. By *registration* we mean determining, in advance, the canonical bit-position within the bit-vector which will represent the registered element. This registration procedure makes sure that, for example, the type specifier `(member nil)` will have exactly the same representation as the type specifier `null`.

An additional problem with `member` is that one could conceivably ask whether a type consisted of exactly one particular element; if the element mentioned just happened to be the representative chosen, our implementation of `subtypep` would give out the wrong answer. Therefore, whenever an element is explicitly mentioned in `member`, and it was previously used as the anonymous representative of other elements, we must somehow generate an alternative element (if one exists) to be used as the new anonymous representative. This procedure will ensure that the type system will not give out wrong answers under these conditions. We will call this situation the *aliasing* problem.

The aliasing problem is relatively easily solved. The aliasing problem cannot happen with numbers, because the interval representation (as discussed later) handles the general case. With most other Lisp object types, the Lisp reader generates a brand-new object which is guaranteed not to alias with any other object of the same type, hence the representative for that type can never become aliased.

From this, we can see that the only possible aliasing problems are with `symbol`'s and `character`'s.[4]

The aliasing problem for symbols can be dealt with by either utilizing a very long symbol name (e.g., `anonymous` itself, or `rumpelstiltskin`), or by utilizing a name guaranteed to be unknown to the caller of `subtypep`—an *un-interned* symbol.

The aliasing problem is the most severe for *characters*, because the set of characters is (normally) finite, and it is conceivable that a user will want to utilize the Common Lisp type system to answer complex questions about sets of explicitly-named characters (e.g., "upper-case", "lower-case", "digit", "alphabetic", etc.). Thus, during the element registration procedure, if a character is registered that was previously the anonymous representative, then another character must be generated that has not yet been mentioned. This requires an *enumerator* for the various subsets of characters: `character`, `base-character`, and `extended-character`, which generates a new character that has not yet been mentioned. Another alternative is to pre-register *all* characters, although this would slow down `subtypep` for the vast majority of questions which did not involve characters at all. A compromise solution would be to temporarily—for a particular call to `subtypep`—register all characters when *any* character is explicitly mentioned.

The example implementation of `subtypep` in Appendix II includes a registration procedure, and therefore handles `member` correctly.

7.3 How to handle numeric intervals

Interval type specifiers such as `(integer 3 (7))` (which means those integers between 3, inclusive, and 7, exclusive; i.e., {3,4,5,6}), cause problems because there is no *a priori* finite set of representatives which could answer all interval questions. Also, it is not at all obvious how to generate an appropriate set of representatives at run-time. Therefore, we will have to develop a different representation from the bit-vector representation we have used so far, in order to handle numeric interval type specifiers.

The general representation for numeric interval type specifiers that we will use is an ordered union of disjoint simple intervals. In our representation, a simple interval is a single clause of the form `(`*ntype low high*`)`, where *ntype* is one of (`integer`, `ratio`, `short-float`, `single-float`, `double-float`, `long-float`) and *low*,*high* are the range end-points.[5] In the ordered union the simple intervals are not allowed to *touch* one another, where *touching* is defined as abutting so closely that the two simple intervals could have been coalesced into one. This representation is closed under the Boolean operations of finite union, finite intersection and complement. All of these operations are essentially *linear* in the size of the unions, since they involve a simple *merging* process for the ordered unions.

There is a slight complication in the definition of "touch" when doing unions in the various number formats. For example, `(ratio 2/3 (1))` *touches* `(ratio (1) 13/2)`, while `(ratio 2/3 (7/8))` does not touch `(ratio (7/8) 13/2)`. Touching is not a property of the *end-points* of a simple interval, but of whether there are any numbers *in between* the two simple intervals. `1` is between `(ratio 2/3 (1))` and `(ratio (1) 13/2)`, but `1` is an `integer`, not a `ratio`, so there are no `ratio`'s between the two sets, thus they *do* touch. `7/8` is between

---

[4]Some Common Lisp readers *coalesce* [CL90,p.694] all quoted expressions, meaning that quoted lists which *print* the same, may actually *be* the same (`eq`); since coalescing is done in a bottom-up fashion, sub-expressions are also coalesced. In such a case, one may have to resort to the "`#,`" notation of Common Lisp to guarantee non-aliased representatives from the reader.

[5]Our representation is a slight generalization of the Common Lisp subrange specifier [CL90,p.61-62], because we do not require that *ntype*$\supseteq${*low*,*high*}, but allow the endpoints to be any orderable numbers; this representation is well-defined due to a recent change in Common Lisp making numeric comparisons precise [CL90,p.290].

`(ratio 2/3 (7/8))` and `(ratio (7/8) 13/2)`, but since `7/8` is a `ratio`, the two simple intervals *do not* touch.

The one complication remaining is the handling of different types of numbers. By keeping different unions of intervals in the 6 different number classes (`integer`, `ratio`, `short-float`, `single-float`, `double-float`, `long-float`)[6] we can handle not only all straight-forward cases, but also all intersections, unions, and complements as well.

Consider the following example. We know that `rational` is the disjoint union of `integer` and `ratio`, so we can represent the simple interval `(rational (2/3) 13/2)` by

`(or (integer (2/3) 13/2) (ratio (2/3) 13/2))`, or after simplification,
`(or (integer 1 6) (ratio (2/3) 13/2))`,

where the integers {1,2,3,4,5,6} are represented in the first set, and the *non-integer* rationals r, $2/3 < r \leq 13/2$, are represented in the second set. Notice that

`(and rational (not (integer 1 6))` becomes
`(or (integer * 0) (integer 7 *) (ratio * *))`,

which means all the non-positive integers, plus the integers greater than 6, plus all non-integer rational numbers.[7]

Using this union of disjoint simple intervals representation, we can canonicalize `member` constructs into equivalent numeric ranges.[8] This simplifies the processing of `member` type specifier clauses. As an example of this process, we transform `(member 3 2/3 3.4s0)` into

`(or (integer 3 3) (ratio 2/3 2/3) (short-float 3.4 3.4))`.

There are some technical problems which arise with floating-point number type specifiers. In order to represent these numbers as numeric intervals (with "open" and "closed" end-points), we have implicitly assumed that floating-point numbers within each floating-point format are *totally ordered*. This means that for any two floating-point numbers x,y of the same format, either x<y, x>y, or x=y; exactly one of these three conditions must hold.

Requiring that floating point numbers be totally ordered has several implications. First, if the underlying implementation supports the IEEE floating point standard [IEEE82], with its "not-a-numbers" ("NaN's"), then the set of supported bit-patterns of the floating-point format may not be totally ordered. For example, a *projective*, or unsigned, infinity, is not totally ordered with respect to the normal floating point values. One result of the existence of these "NaN's" is that the type specifier `single-float` is no longer identical to the type specifier `(single-float * *)`, because the first contains the NaN's, while the second contains only the finite floating point numbers. (Note that the IEEE NaN's $+\infty$ and $-\infty$ are also not included in `(single-float * *)`.)

---

[6] In the case where one or more of the differently named floating-point types are actually identical ("aliased"), we map the named type into the representational type. For example, `(short-float * *)` may map into `(single-float * *)` during canonicalization if there is no separate `short-float` type.

[7] In our representation of rational ranges, we have assumed that `rational≡integer∪ratio`, as in most Common Lisp implementations. Common Lisp may allow for implementations with rationals which are not integers or ratios [CL90,p.39], but standard mathematics requires that these other numbers *act* like integers and ratios—i.e., they can be compared using <,=,>,etc., and arithmetic can be performed using +,-,*,etc. Our scheme ignores the differences between these "fake" integers/ratios and "real" integers/ratios, and treats them both as if they were "true" integers and ratios. In this sense our scheme ignores the *representation* of a number and focuses on its *value*.

[8] This canonicalization relies on `type-of` [CL90,p.65-67] to produce the correct number type for labelling the single-point simple "interval". This usage neatly finesses the problem of "aliased" floating-point types, where an implementation may call a single floating-point type both `short-float` and `single-float`.

Second, the total order on floating-point numbers must be consistent with `eql`, rather than =, else the canonicalization of `(member x)` into `(single-float x x)` is not valid. This is because `member` is defined in terms of `eql` [CL84,p.44], while intervals are defined in terms of <, >, and =. The only *numeric* value this should cause trouble for is *minus zero* (`-0.0`).[9] `-0.0` causes real headaches because it cannot be distinguished from `0.0` by the numeric comparison predicates (=, <, >, etc.), but it *can* be distinguished by `eql` [CL84,p.79]. The determination of the numeric intervals such as `(float 0.0 1.0)` is performed by the standard numeric ordering predicates (<, <=, etc.), while `(member -0.0)` is defined to use `eql` for its determination [CL84,p.44]. Our canonicalizing procedure above would represent `(member -0.0)` as `(single-float -0.0 -0.0)`, which is exactly the same set as `(single-float 0.0 0.0)`—i.e., {`-0.0`,`0.0`}—because the numeric ordering predicates cannot distinguish between `0.0` and `-0.0`. In other words, our representation—as presented so far—cannot handle the singleton set {`-0.0`}. The simplest way to fix this problem is to define a new set of ordering predicates (<', <=', etc.), which *can* distinguish between `0.0` and `-0.0`, and then to use these predicates in the internal workings of our ordered union operations. When canonicalizing using these predicates, we must be careful to fix up intervals which came in from *outside* `subtypep` and involve `0.0` as an end-point before further processing. This is necessary since these intervals implicitly include `-0.0` due to the lack of resolution of the standard comparison functions. This "fixup" would add `-0.0` to the interval; for example, converting `(float 0.0 1.0)` into `(float -0.0 1.0)`. (Needless to say, many computer scientists wish that `-0.0` would go quietly away.)

Thus, to handle floating-point numbers correctly, we must handle NaN's as a discrete type separately from the normal floating point numeric ranges (i.e., `(and single-float (not (float * *)))` contains the `single-float` NaN's and is treated as an elementary type disjoint from `(single-float * *)`), while within the numeric ranges we use an ordering predicate which is consistent with `eql`.

So far we have talked only about `rational` and `float` numbers, but not about `complex` numbers. It was not clear in Common Lisp-84 whether type specifier expressions such as `(complex fixnum)` or `(complex (integer -10 10))` were legal, and if legal, how they were to be interpreted. The only requirement was that both the real and imaginary parts of a complex number were of the same type [CL84,p.19]. Common Lisp-90 provides a new function `upgraded-complex-part-type` [CL90,p.68], which legitimizes and defines type specifiers such as `(complex (integer -10 10))`. If the range of the function `upgraded-complex-part-type` is the set {`rational`, `short-float`, `single-float`, `double-float`, `long-float`}, then the various subsets of `complex` can be easily handled by a small number of representatives and treated as discrete types. If, however, the range of this upgrading function includes subranges of these types, then in order to properly implement a decision procedure for `subtypep`, we are required to represent unions of *rectangles* on the complex plane, and while this is straight-forward, it is very tedious. (Note, however, that similar algorithms already exist in most computer graphics "window systems".)

The Common Lisp non-complex *numbers* are thus seen to be the Cartesian product of the 6 numeric types mentioned above, where each numeric type can be represented by an ordered union of disjoint simple intervals. Once we can put the simple range type specifiers into a canonic form which is an ordered union of disjoint simple intervals, we can then combine them using `and` (intersection), `or` (union) and `not` (complement). It is then trivial to test it for emptiness, i.e., determine `type-null`.

7.4 Handling arrays

Type specifiers for array types utilize the following form: `(array` *element-type bounds*`)`, where *element-type* is another type specifier expression and *bounds* is a form specifying what information is known about the bounds of this array. Specifiers such as `string` and `bit-vector` are expanded

---

[9]The various infinities are not numeric values.

into `(array string-char (*))`[10] and `(array bit (*))`, as if they had been defined by `deftype`. We actually use an internal form for array type specifiers which incorporates "simplicity" (`simple`/non-`simple`) information as another component; this complication adds additional complexity but no additional insight.

The type specifier `(array` *element-type bounds*`)` can actually be canonicalized into `(array` *x bounds*`)`, where *x* is the "smallest" of the finite number of array-element-types that the implementation provides which is large enough to hold *element-type* (here "smallest" and "large enough" refer to `subtypep` recursively). *Element-type* can also be the lexical element "*". In this case the expression represents the union of array types `(array` *x bounds*`)`, where *x* ranges over all of the array-element-types that the implementation supports.

The bounds specifier can be "*", meaning no information, "*n*", meaning all rank-*n* arrays, or "(*m n...p*)", meaning all arrays with bounds (*m n...p*). Finally, one or more of the *m*, *n*, *...p* can be "*", meaning "no information for this dimension". Clearly, `(array` *x* `*)` is the countable union of `(array` *x i*`)`, for all non-negative integers *i*, `(array` *x* `(*))` is the countable union of `(array` *x* `(`*j*`))`, for all non-negative integers *j*, and `(array` *x* `(4 *))` is the countable union of `(array` *x* `(4` *j*`))`, for all non-negative integers *j*, etc.

A decision procedure for array type specifier expressions is complicated by the fact that `subtypep` can ask some difficult questions about arrays, such as:

```
(subtypep '(array short-float (3 4))
          '(and (array * (* 4)) (array single-float (3 *))))
```

The answer is *true* for an implementation in which `short-floats` and `single-floats` have the same representation. The question is complicated by whether the implementation utilizes different type *containers* in the different arrays, and also whether `subtypep` is smart about array bounds.

We first note that an implementation usually implements arrays of a finite number *k* of *element-type*'s and maps all other arrays into those *k* different kinds of arrays. Common Lisp requires *k* to be at least 3—general arrays, character arrays, and bit-arrays [CL84,p.29]). We will treat each of the *k* implementation-dependent array-types similarly but separately. Within each of those array types, we need to handle each of the different possible array-bounds cases.[11]

In order to handle the different possibilities of array-bounds, we must choose representatives for every array *rank* (at least up to `array-rank-limit`), and then within that rank, choose appropriate representatives. If Common Lisp allowed only type specifiers such as: `(array * 3)` or `(array * *)`, then rank differences would be the only issue, and we could choose our representatives in advance: associate the integer 2^i with rank i, and the integer -1 with rank "*". Under this scheme, using two's-complement integers, -1 would represent *all* of the different ranks $[0,\infty)$, while each particular rank would be associated with a positive power of 2. This preselection strategy for representatives will not work, however, for those more complex questions allowed by Common Lisp, an example of which was shown above.

We are thus led to a strategy by which appropriate representatives are chosen after a particular type specifier expression is examined. We will continue to utilize integers (as infinite bit-vectors) to

---

[10]Although `string-char` is no longer a defined Common Lisp type [CL90,p.460], we here use "`string-char`" to mean the union of possible string element types. We could alternatively expand "`string`" as "the union of [the] one or more specialized vector types" that are used to implement strings [CL90,p.460].

[11]Some Common Lisp array implementations may remember the precise actual array element type that was used when the array was created—i.e., the function `upgraded-array-element-type` produces a canonical representation of the given argument type. Such an implementation would appear to have an infinite number of different `array-element-type`'s. In this case, `subtypep` must keep track of the `array-element-type`'s actually mentioned, as we do for the different dimensions, rather than attempting to canonically represent all possible `array-element-type`'s.

represent array bounds, but we will allocate more than one bit position for any array bounds whose integer indices are *mentioned* in the type specifier expression; e.g., (3 4), (3 *), and (* 4), above. In the particular case above, we will require a single representative for the bounds (), (*), (* * *), (* * * *), etc., but we will require the following 4 representatives for the rank-two bounds: (3 4), (3 @), (@ 4), and (@ @). We use "@" to represent the "@nonymous" representative of the unmentioned bounds.

We find that only (3 4) belongs to the set represented by the specifier "(3 4)", but both (3 4) and (3 @) belong to the set "(3 *)", both (3 4) and (@ 4) belong to "(* 4)", and all four elements belong to "(* *)". In other words, array bounds specifiers like "(3 *)" represent *slices* through a multidimensional rectangular box of bits, each of which is a particular representative for the array bounds mentioned in its coordinate position. The process of choosing representatives and allocating bit-positions is similar to, but much more complicated than that for *registering* elements in a `member` construct. In general, the number of bit positions required for rank r is equal to $(d_1+1)(d_2+1)...(d_r+1)$, where the d_i are the number of distinct integers mentioned in dimension i (not counting "*"). In the case where no specific dimensions are mentioned, the formula gives 1 bit position, which is equivalent to the unmentionable bounds (@ @ … @).

In the example above, we will need 1 bit position for arrays of rank 0, 1 bit position for arrays of rank 1, 4 bit positions for arrays of rank 2, and 1 bit position for arrays of rank 3 and above. Rank 0 is mapped into bit position 0, rank 1 is mapped into bit position 1, rank 2 is mapped into bit positions 2-5, rank 3 is mapped into bit position 6 and so forth for higher ranks. For the row dimensions, we assign "3" the index 0 and "@" the index 1; for the column dimensions, we assign "4" the index 0 and "@" the index 1; we utilize the same row-major order as Common Lisp. Thus, the encoding for `(array * (3 4))` (ignoring the element-types) is `#B00000100` or `+4`; the encoding for `(array * (3 *))` is `#B00001100` or `+12`; the encoding for `(array * (* 4))` is `#B00010100` or `+20`; the encoding for `(array * (* *))` is `#B00111100` or `+60`. Since `(logand 12 20)=4`, the two type specifier expressions are equivalent (not considering element-type and simplicity). (Notice that `(array * 3)` is encoded as 2^6=64, while `(array * *)` is encoded as -1.)

We do not even require the use of an entire integer for our bit-vector, since for any given type specifier expression, only a finite number of different array ranks are actually mentioned. If we allocate 1 more bit position than the highest rank actually mentioned, then we will be sure to include at least one representative for all the higher-ranked arrays. Using this technique, we can allocate the array bounds bit-vectors for all different specialized element-types and simplicities as subsequences of a single bit-vector. Furthermore, this array bit-vector could be a portion of the bit-vector for the basic non-array type information.

Handling the `member` construct with elements which are array objects is somewhat problematical. Unless a type specifier is constructed at run-time, it is impossible to ask a non-trivial `subtypep` question about array objects. This is due to the fact that the types of arrays created by the reader are defined by Common Lisp [CL84, p.346-357] as being simple vectors of element-types `bit`, `string-char` or `t`, or simple arrays of element-type `t`.[12] However, the simulation of `typep` using `subtypep` above requires the ability to handle any type of object within a `member` construct, including array objects.

Handling actual array objects in a `member` construct is possible using our algorithm by increasing the number of bits within the subsequence of the bit-vector which are allocated to the actual bounds of the given array. In other words, if an array of actual dimensions (3 4) were mentioned in a `member` construct, then we would need to allocate at least two bits for arrays of dimension "(3 4)" in the bit-vector; one bit for this specifically mentioned array, and one bit for all the anonymous, un-

[12] The one type of non-trivial question one could ask is `(type-equal (member "a") (member "a"))`, i.e., does the reader share constant strings or not? This issue was discussed before in the footnote on the "`member`" section.

mentioned arrays of the same dimensions. However, for non-`simple` arrays, the actual dimensions may not be well-defined, because they can be changed by `adjust-array` if they are `adjustable`. (This is the only situation within Common Lisp-84 [CL84] where an object can change its type and remain `eq` to itself—a highly undesirable situation.)

For these reasons, we recommend against including array objects as elements in a `member` construct, or trying to handle this case in a more algorithmic manner, until Common Lisp defines the semantics more completely.

7.5 Handling user-defined data types

There are three kinds of user-defined data types in Common Lisp—`deftype` types, `defstruct` types and `defclass` (CLOS) types. `deftype` types are easily dismissed as type specifier "macros" which are expanded every time they appear, and expand into non-recursive type specifier expressions. Thus, `deftype` types are used for brevity only, and do not require much additional machinery to handle properly. In this manner, many Common Lisp built-in type specifiers can be treated as `deftype`'s—`bit`, `(mod` *n*`)`, `(signed-byte` *s*`)`, `(string` *size*`)`, etc.—an approach used by many Common Lisp implementations.

`defstruct` types are so called, because they are defined by the user with the `defstruct` construct, which defines a new "structured" type with components (like a *record* in other languages). If one `defstruct` type "extends" another by including all of its components, then the second type is considered a subtype of the first. Objects for a particular `defstruct` type, also called "instances" of the `defstruct` type, are normally created using an automatically-defined, specialized function `make-<typename>`. CLOS types are those types added by CLOS as new classes are defined by `defclass`. For the purposes of our decision procedure, `defclass` types act analogously to `defstruct` types.

`defstruct` and `defclass` types are more troublesome than `deftype` types, because they materially extend the Common Lisp type system at run-time. As a result, they introduce some ambiguities in the interpretation of `subtypep`. For example, if `foo` is a `defstruct` type, and no instances of `foo` have yet been created using the function `make-foo`, what should be the answer to the question `(subtypep 'foo nil)`? At the time the question is asked, the answer is yes (`t`), but once an instance has been created, the answer should be no (`nil`). Unfortunately, if one requires `subtypep` to accurately answer any such question regarding user-defined type elements, `subtypep` would be required to be able to enumerate all elements of the given type. This process could require a very long time, and would be unnecessary in all but the most perverse circumstances.

It is unreasonable to expect Common Lisp's `subtypep` predicate to register as representatives every instance of a `defstruct` datatype ever created. Yet without such a registration procedure, `subtypep` will never be able to properly answer all possible questions about those objects. This is because the user can always remember each created instance himself, and then ask—utilizing `member`—whether the list of instances was equal to the entire set; `subtypep` would have to remember all the instances itself in order to answer correctly. However, if we change the interpretation of user-defined data types slightly, we can correctly answer all questions regarding user-defined data types.

We propose that `defstruct` itself always create at least one "dummy" anonymous instance of the data type. Since the user cannot become acquainted with this "dummy" instance, he will never be able to ask a `type-equal` question, but only a type containment question. Every user-defined datatype is then non-empty at birth, since the "dummy" anonymous element is created and registered with the type system. Yet since every call to `make-xxx` creates a brand-new instance never before seen, the user can never ask the precise question about exactly which instances are elements of the type.

This policy is reasonable, since it is already followed for the built-in types—it would be most unusual if a Common Lisp system did not create at least one element of each of the built-in datatypes at initialization or system construction time. (We have already seen a similar problem with `symbol`'s—we need a symbol not mentioned or mentionable by the user—hence the `rumpelstiltskin` symbol.) Therefore, the existence of anonymous elements of user-defined datatypes which the user cannot name should not be unsettling.[13]

Another interesting question arises from the need of all Lisp systems to perform garbage collection. Suppose that all instances of `foo` are garbage-collected; will `(subtypep 'foo nil)` now answer true? If a "dummy" instance of `foo` is still held by the type system, then it will not be garbage-collected, and the answer will be "no". This interpretation is equivalent to saying, "yes, the type `foo` has elements, but they are not accessible". Since the determination of accessibility is extremely expensive, requiring an immediate full garbage collection, a more efficient, and hence more useful `subtypep` is obtained using the simpler interpretation of including inaccessible elements in the type.

Handling `defclass` (CLOS) types is even easier for `subtypep` than handling `defstruct` types.[14] This is because CLOS is more explicit about the meaning of classes, as well as the mapping of classes into types. A class cannot be `type-equal` to one of its superclasses or subclasses [CL90,p.774], and CLOS goes on to require certain additional answers to `subtypep` questions where certain built-in types are concerned. The fact that a class cannot be `type-equal` to a sub- or super-class, guarantees that our technique of always registering a single dummy element for each different class will always work.[15] That dummy element serves as a distinguishing element which distinguishes that class from every other class.[16]

Our decision procedure does not handle certain cases where CLOS is used to extend the Common Lisp-84 primitive datatypes, specifically numbers and arrays, which `subtypep` handles specially. Let us first consider the problem of extended numbers. If the numeric type which is being extended has the *metaclass* `built-in-class`—the normal case—then any extension is an error [CL90,p.781]. If the numeric type has the metaclass `standard-class`, then the new type will be a proper subtype of its parent, which means that any numeric operations applicable to the parent type will be applicable to the new type. This is because Common Lisp numeric operations are *not* "generic" [CL90,p.1024], and therefore cannot be specialized to operate differently for this new type. However, this lack of specialization applies also to slots. Since the numeric operations are

---

[13]Some `defstruct` types are not intended to have instances, but are provided for the convenience of two or more other `defstruct` types which "include" the first. Since Common Lisp does not provide any mechanism for a programmer to enforce this intention, our scheme is consistent.

[14]Note that in this paper, we show how to implement `subtypep`, not `typep`. This means that we depend upon a correct implementation of `typep` for the proper behavior of `subtypep`. For CLOS classes, the implementation of `typep` might be quite involved and complicated. Some CLOS implementations might even find it advantageous to implement `(typep x y)` as `(values (subtypep (type-of x) y))`, since the `subtypep` implementation described here already provides the machinery to answer the hard questions. However, the proper implementation of `typep` is outside the scope of this paper.

[15]CLOS defines `(subtypep c d)` ≡ d∈cpl(c), where cpl(c) is the "class precedence list" of c [CL90,p.774,780-781], i.e., cpl(c) ≡ {x∈Classes | `(subtypep c x)=t`}. However, `(subtypep c d)` if and only if d∈cpl(c) if and only if cpl(c)⊇cpl(d). Since the bit-vectors for c,d encode information equivalent to the sets cpl(c),cpl(d), our comparing of the bit-vectors is formally equivalent to the CLOS definition.

[16]Unlike the situation with structures, a Common Lisp programmer can (obliquely) express his intention that no direct instances of a particular class be created, by overloading `make-instance` to signal an error. In such a case, our dummy element is a bastard. Since CLOS does not allow any subclass of this class to be `type-equal` to the class, even though the sets of (direct and indirect) instances may in fact be the same, CLOS thus destroys the extensionality of types. [McAllester86] calls classes of this kind *uninstantiable classes*. We believe that CLOS should allow a programmer to assert that a class is `uninstantiable`. We discuss this issue again in Section 9.

not specialized, they cannot access any slots specific to the new type, and therefore their behavior will be exactly the same as with instances of the parent type—i.e., "normal" numbers. The instances of the extended type will be numbers with some "hidden" slots which only we know about, and since most arithmetic operations will generate a new "normal" number, these hidden slots will not be propagated. These "extended numbers" are thus essentially useless.

A more interesting possibility is that posed by extending the `array` type/class. Since the standard array manipulating functions are not "generic" [CL90,p.1024], we would have the same problem that we had with extended numbers—the behavior of these functions could not be specialized to the extended type. However, array manipulations tend to create new array instances much less frequently than numeric calculations create new numbers, so the lack of propagation of any extended type-specific slots would not be a problem. The major problem for such an extended type is how an instance could be created. `make-array` could not be used, since it would create a normal array type. Since the array type is extensible, it is presumably an instance of `standard-class`, in which case there is an overloaded definition of `make-instance` which works for this type, which can be used to overload `make-instance` for the extended type. Extending our `subtypep` decision procedure to handle such array types could be quite difficult.

As a result of these observations, we expect that most CLOS implementations will have all numeric and array types "built-in", and hence incapable of extension/subclassing. In such implementations, numbers and arrays will then be disjoint from all other CLOS classes, as well as from each other, and so our "divide and conquer" strategy for dealing with numbers and arrays will continue to work.

There is one other aspect of CLOS which is quite troublesome for our type system. This is the possibility of an object *changing its type* dynamically during run-time. With the sole exception of non-simple arrays, which can change their array bounds (and hence their type) dynamically, every other Common Lisp-84 object has a static type—i.e., the type of an object does not change after creation. The CLOS function `change-class` and the macro `defclass` can both modify the type system in such a way that an object's type can change dynamically. Both operations have the possibility of inflicting major damage on the type system, and are therefore quite dangerous. Given the major uses of `subtypep` in Common Lisp—for declarations and storage allocation—it appears that much is lost through these dynamically changing types.

7.6 Handling SATISFIES

The existence of `satisfies` clauses in a type specifier expression could be considered sufficient grounds for `subtypep` answering "can't determine". We can handle many of the easier forms of `subtypep` involving `satisfies`, however, if we first expand the argument type specifier expressions in the manner indicated in the "obviously trivial" version of `subtypep` previously described in section 5. If this expansion is performed in the proper order, `subtypep` should at least be able to answer trivial questions such as:

```
(subtypep '(and integer (satisfies oddp)) 'integer).
```

While this is not a decision procedure, this set of heuristics should be capable of handling many of the easier questions; it may also slightly speed up questions not involving `satisfies`.

One could also extend the basic (non-`satisfies`) decision procedure to handle certain cases of `satisfies` as follows. Treat each predicate symbol that occurs in a `satisfies` clause as an *uninterpreted* type name, and use a decision procedure for uninterpreted monadic predicate symbols [Church56] [Rackoff75] to decide the `subtypep` question. Such an extended `subtypep` decision procedure would then be well-defined, and would answer "can't determine" only when the answer would depend upon the specific interpretation of the predicate symbol—i.e., the actual semantics of the predicate. This extended `subtypep` decision procedure would then be NP-complete, since we could use it to do tautology-testing.[17] While strictly more powerful than the non-extended decision procedure, we do not recommend this approach, since the additional machinery could slow down *every* `subtypep` decision, even when it did not involve `satisfies`.

7.7 Putting it all together

In order to decide a `subtypep` question, we divide the whole question into several sub-questions by computing the homomorphic image of the type-specifier expressions for each of the kingdoms. For example, the homomorphic image of `(or symbol (member 3) bit-vector)` is `(or symbol nil nil)≡symbol`, `(integer 3 3)`, or `(array bit *)`, etc., depending upon the kingdom being decided. Each of these subquestions is then passed in turn to decision "experts" for each of the main type "kingdoms" of Common Lisp: flat or discrete types, numeric ranges, array specifiers, and user-defined types. The answer to the whole question is only true if *all* of the kingdom experts answers true. This conjunction results from the nature of the Lisp datatype Boolean algebra as the Cartesian product of the disjoint kingdoms.

Thus, we have shown how any `subtypep` question without the `satisfies` construct can be answered by building a constructive model of the types (= sets) involved, and then performing Boolean operations on those sets. It has also been shown that, if ranges are not involved, the representation of any type specifier expression can be a single binary integer, and that any subtype question can be answered with a single logical bit-vector computation. The size in bits of these integers is relatively small in practice (perhaps 50 bits), although in the worst case, they can grow exponentially in the size of the type specifier expression (due to the explosion of bits required for complex expressions involving higher-order arrays).

The independent computation for each of the classes of types: basic types, ranges and arrays, are all performed in parallel (hence the ability to implement most of the work using bit-strings). Thus, the worst case for a *meet* or a *join* operation is the logical *and* or *or* of several bit strings, along with a number of *range-meet* or *range-join* operations for each of the different numeric range types.

The code to implement this `subtypep` decision procedure, in addition to being fast, is also quite small, typically requiring fewer lines of code than the `subtypep` that is most likely provided with most implementations (e.g., 3 pages for Kyoto Common Lisp's `subtypep`, 15 pages for Symbolics Common Lisp, 18 pages for Spice Common Lisp's `subtypep`). Furthermore, it works on even the most complicated type specifier expressions, eliminating the need to analyze some very complex code (e.g., `hairy-subtypep` and friends).

As a side-effect of this effort, one also gets the lattice operations of `meet` (= `and`) and `join` (= `or`) for the Common Lisp type system, which can be used for type inference [Baker90]. Additionally, one gets a very interesting representation and manipulation package for range arithmetic that can handle ranges with holes; e.g., the domain of the reciprocal function 1/x. Unfortunately, `meet` and `join` produce their answers in a highly stylized canonical form, and converting from this form back into a readable type specifier expression produces very complex and hard-to-understand type specifier expressions. One can attempt to produce more readable expressions, but producing the shortest (or simplest?) type specifier expression equivalent to a particular canonical form is an NP-complete problem.

---

[17]McAllester's boolean class system [McAllester86] utilizes a similar system to determine whether a certain class in an intentionally specified class system is allowed to have any members. His classes are defined by (essentially) uninterpreted monadic predicates among which certain subset relations are posited.

8. Complexity

The time and space required to execute the decision procedure described above can be determined by examining each of the components. We will measure complexity relative to the size of the arguments to `subtypep`. The basic operations are as follows:

- splitting the problem into disjoint domains
- solving the problem for each domain
- putting the answers together

Splitting the problem into disjoint domains requires work linear in the size of the input argument, and putting the results back together is a simple conjunction whose size is proportional to the number of kingdoms. This splitting leaves most of the complexity within the domain specific decisions.

Discrete types are the built-in elementary types, excluding numeric ranges and arrays. We show in Appendix II an implementation of a decision procedure for discrete types which utilizes perhaps 30 representatives for the built-in Common Lisp discrete types. Decisions involving `member` can cause the bit vectors to grow, but still in proportion to the *elements* mentioned.[18] The actual set operations with the bit-vectors will be of approximately *quadratic* complexity, since both the number of operations, as well as their size, will be proportional to the size of the input argument. The registering procedure itself can take anywhere from constant time per element registered to a time proportional to the total number of registered elements; this will be implementation dependent. This implementation dependency arises because during the registration of an element, we must call `typep` for each discrete type known to the system, so that their bit-vectors can be updated.

Numeric range decisions generally involve the manipulations of ordered unions of intervals, where the size of the unions is relatively small. However, one could construct examples where the unions attain lengths which are exponential in the size of the input argument. These cases are highly artificial, however, because they involve continually subdividing intervals into smaller and smaller pieces. In the typical case, the unions are extremely short—generally one or two intervals.

Array type decisions involve the manipulations of relatively short bit-vectors. However, one can construct artificial examples where the lengths of the bit-vectors are exponential in the size of the input argument. A certain amount of this complexity is forced, since we can ask the sort of database retrieval question posed in section 6 using array types. However, we consider such examples artificial, since they arise from arrays of high rank and widely varying dimensions.

User-defined type decisions (without `member`) involve the manipulation of a number of representatives which is proportional to the number of user-defined types. Note, however, that any work in registering these representatives is not charged to `subtypep`, because these types cannot be defined within a call to `subtypep`. While the amount of work involved in registration might be substantial, charging it to the creation of the type rather than to `subtypep` is reasonable. (The work of registering representatives is also likely to be completely swamped by the other activities involved in the creation of a type.) If there are a large number of built-in types, the normal expectation is that once defined, these types remain relatively constant, while `subtypep` could be called a large number of times. Thus, the amount of work charged to `subtypep` is at most a large constant times the size of the input specifier.

Including actual objects in a `member` construct will cause the registration of those objects to be charged to `subtypep`, however. In such cases, a significant amount of work may have to be done if there are a large number of user-defined types, as the work to register is proportional to the number of user-defined types.

---

[18]The set of representatives need grow only as the *logarithm* of the number of types, since n representatives can distinguish 2^n different types.

In summary, the typical user of `subtypep` will see its execution taking an amount of time linearly proportional to the size of the input specifiers, while heavy users of `member` and user-defined types will experience a more quadratic performance. Only the most perverse user will experience the exponential worst case behavior of this algorithm.[19]

9. Implications for future type systems

The decision procedure developed for Common Lisp depends critically upon the exact language of type specifiers. Changes in the type system in the direction of more specificity or changes in the combining operators allowing more generality would have a profound effect on the ability of a mechanical procedure to decide `subtypep` questions. For example, the equivalence of functional types is undecidable, so any change in the direction of Scheme [Rees86] for allowing the comparison of functions will make the problem intractable for Common Lisp.

On the other hand, Common Lisp's use of `eql` semantics for the `member` type specifier makes many problems trivially decidable—for example, those of "equivalent" functions. Whether `eql` semantics is the most practical semantics within `subtypep` is not known.

The attempt by CLOS to overload the predicate `subtypep` with the meaning "subclassp" is quite unfortunate. Common Lisp already has a precise meaning for `subtypep`—extensional subset—while CLOS attempts to convert it into a predicate about intensions. This CLOS meaning destroys the possibility of building *abstract* data types whose implementation is hidden, because the ability to test for "subclassp" means the ability to determine the superclasses of a class, and hence the slots of the class. The CLOS meaning thereby confuses membership ("is-a") with implementation ("composed-from"). For example, one should not be able to deduce that a "set" is a "sequence" just because one happens to use a "sequence" to implement a "set". We suggest that CLOS implement instead a predicate `subclassp`, and base method selection on this predicate, while allowing `defclass` to independently specify extensional inclusion of the generated type.

We also suggest that Common Lisp provide an "uninstantiable" specification for user-defined types which specifies when a new type is not allowed to have any direct instances, so that the intention of user-defined types which are partitioned into subtypes which have direct instances can be specified. This terminology is introduced by McAllester in his very interesting paper [McAllester86] which provides a consistent interpretation for inheritance in a class system which has been completed into a Boolean algebra.

The current Common Lisp type specifier system does not provide the best possible information regarding types in declarations for use in highly optimizing compilers. For example, one cannot specify ranges which are bounded by a run-time variable in Common Lisp. Such type information is highly useful in Ada, for example, for removing redundant run-time range checks [Ada83]. Common Lisp curiously does not provide descriptive information about the types of items in its list structures, yet this information can be invaluable in producing highly efficient code for manipulating lists [Milner78]. Unfortunately, neither of these capabilities can be handled by the decision procedure described here.

However, there are a few type specifier changes which *could* be handled by our techniques. One of these is the ability to describe the "alignment" of an integer as being divisible by 2, 4, 8, etc. Such information would be extremely valuable in compiling the most efficient code for RISC architectures which can access objects only on certain address alignments.

The current Common Lisp type system does not provide for any kind of *context* marker for the interpretation of `typep` and `subtypep` questions. Such a context marker would be very helpful in

---

[19]This exponential behavior is caused only by the perverse use of arrays. Unlike [McAllester86], in which certain questions are co-NP complete because questions are asked about *all* interpretations of a system of subsets, our procedure is fast because it is asking about a *particular* system of subsets.

distinguishing between compile-time and run-time type environments, especially where user-defined types are involved. The lack of such a context marker makes the algorithm for deciding `subtypep` a lot easier, but a lot less useful. This is because the important issue within a compiler is what values a object could take on in the environment of the run-time system, not what values it can take on in the compile-time environment. As a result of this omission, Common Lisp compilers will almost certainly utilize a different `subtypep` than the standard run-time version, and the compile-time version will have to be less precise in order to allow for the differences between the compile-time and the run-time environments.

10. Conclusions

Logicians have long known that two logical systems related to the Common Lisp type system are known to be decidable: the first-order theory of uninterpreted sets [Church56] [Rackoff75], and the first-order theory of rational order [Ferrante77] [Ferrante75] (a much easier version of Tarski's theory of *real closed fields* [Tarski51]). It is almost obvious that these two results make the Common Lisp type system (without `satisfies`) decidable. However, both of these decision procedures have multiply-exponential time bounds, and are of theoretical interest only.

The decision procedure described here is similar to that for the first-order theory of uninterpreted sets in that it builds a constructive *model* using objects gleaned from the input arguments. This procedure is very much cleaned up, however, for efficient implementation. Our decision procedure for intervals is new, although it is a reasonably obvious extension of standard interval arithmetic. It has not been described before most likely because programming it in Fortran is too painful to contemplate.

Acknowledgments

Jeff Hirsch provided invaluable help in editing earlier versions of this paper. We are also indebted to Carolyn Talcott and Richard Gabriel who refereed this paper and gave many valuable criticisms.

References

Ada83. *Reference Manual for the Ada® Programming Language*. ANSI/MIL-STD-1815A-1983, US GPO, Wash, DC, 1983.

Aho86. Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

Baker90. Baker, Henry G. "The Nimble Type Inferencer for Common Lisp-84". Submitted to *ACM Trans. on Prog. Langs. and Systems.*

Bauer74. Bauer, Alan M., and Saal, Harry J. "Does APL really need run-time checking?" *Software Practice and Experience*, v.4, 1974,pp.129-138.

Beer88. Beer, Randall D. "The compile-time type inference and type checking of Common Lisp programs: a technical summary". TR 88-116, Ctr. for Automation and Intelligent Sys. Research, Case Western Reserve Univ., May 1988.

Bobrow88. Bobrow, et al. "Common Lisp Object System Specification X3J13", *ACM SIGPLAN Notices*, v.23, Sept. 1988; also *Lisp and Symbolic Computation 1*, 3-4, pp245-394; also X3J13 Document 88-002R, June 1988.

Borning82. Borning, Alan H. and Ingalls, Daniel H. H. "A Type Declaration and Inference System for Smalltalk" *ACM POPL 9*, 1982, pp.133-141.

Budd88. Budd, Timothy. *An APL Compiler*. Springer-Verlag, NY, 1988.

Church56. Church, A. *Introduction to Mathematical Logic, Vol. I.* Princeton University Press, 1956.

CL84. Steele, Guy L., Jr. *Common Lisp: The Language*. Digital Press, 1984.

CL90. Steele, Guy L., Jr. Common Lisp: The Language, Second Edition. Digital Press, Bedford, MA, 1990.

Ferrante, J., and Rackoff, C. "A decision procedure for the first order theory of real addition with order". *SIAM J. Comput. 4*, 1 (1975),69-76.

Ferrante, J. and Geiser, J. "An efficient decision procedure for the theory of rational order". *Theor. Computer Sci. 4*, 2 (1977),227-234.

Hughes68. Hughes, G.E., and Cresswell, M.J. *An Introduction to Modal Logic*. Methuen and Co., 1968.

IEEE82. Floating-Point Working Group 754 of the Microprocessor Standards Committee. "A Standard for Binary Floating-Point Arithmetic". IEEE, 345 E. 47'th St., New York, NY 10017, 1982.

Loos83. Loos, R. "Computing in Algebraic Extensions". In Buchberger, et al. *Computer Algebra: Symbolic and Algebraic Computation, Second Edition*. Springer-Verlag, New York, 1983, pp.173-187.
Kaplan80. Kaplan, Marc A., and Ullman, Jeffrey D. "A Scheme for the Automatic Inference of Variable Types". JACM 27,1, Jan. 1980, pp.128-145.
Ma, Kwan-Liu, and Kessler, Robert R. "TICL—A Type Inference System for Common Lisp". *SW—Prac.&Exper. 20*,6 (June 1990),593-623.
MacLane67. MacLane, Saunders and Birkhoff, Garrett. *ALGEBRA*. Macmillan, 1967.
McAllester, David, and Zabih, Ramin. "Boolean Classes". *Proc. '86 OOPSLA, Sigplan Notices 21*,11 (Nov. 1986),417-423.
Milner78. Milner, Robin. "A Theory of Type Polymorphism in Programming" *JCSS* 17, 1978,pp.348-375.
Morris73. Morris, J.H. "Types are Not Sets". *ACM POPL*, 1973, pp.120-124.
Rackoff75. Rackoff, C.W. "The complexity of theories of the monadic predicate calculus". IRIA Rep. 136, Roquencourt, France, 1975.
Rees86. Rees, J. and Clinger, W., et al. "Revised Report on the Algorithmic Language Scheme". *SIGPLAN Notices 21*, 12 (Dec. 1986), 37-79.
Scott76. Scott, D. "Data types as lattices". *SIAM J. Computing*, 5,3 (Sept. 1976), 522-587.
Steele78. Steele, Guy L., Jr. *Rabbit: A Compiler for SCHEME (A Study in Compiler Optimization)*. AI-TR-474, Artificial Intelligence Laboratory, MIT, May 1978.
Suzuki81. Suzuki, Norihisa. "Inferring Types in Smalltalk". *ACM POPL 8,* 1981,pp.187-199.
Tarski51. Tarski, A. *A Decision Method for Elementary Algebra and Geometry*, 2nd ed., Univ. of Cal. Press, Berkeley, 1951.

Appendix I. Axioms for Lattices and Boolean Algebras [MacLane67].

A *lattice* is a set U together with a predicate $\leq$ on U.

For all x, $x \leq x$.	(Reflexivity)
If $x \leq y$ and $y \leq x$, then $x = y$.	(Antisymmetry)
If $x \leq y$ and $y \leq z$, then $x \leq z$.	(Transitivity)

Define $x \wedge y$ to be *greatest lower bound* of x,y under $\leq$. Define $x \vee y$ to be *least upper bound* of x,y under $\leq$.

$x \wedge x = x$, $x \vee x = x$	(Idempotency)
$x \wedge y = y \wedge x$, $x \vee y = y \vee x$	(Commutativity)
$x \wedge (y \wedge z) = (x \wedge y) \wedge z$, $x \vee (y \vee z) = (x \vee y) \vee z$	(Associativity)
$x \wedge (x \vee y) = x \vee (x \wedge y) = x$	(Absorption)
$x \wedge y = x$ if and only if $x \vee y = y$ if and only if $x \leq y$	(Consistency)
If $y \leq z$, then $x \wedge y \leq x \wedge z$ and $x \vee y \leq x \vee z$	(Monotonicity)
$x \wedge (y \vee z) \geq (x \wedge y) \vee (x \wedge z)$,	(Distributive *in*equalities)
$x \vee (y \wedge z) \leq (x \vee y) \wedge (x \vee z)$	
If $x \leq z$, then $x \vee (y \wedge z) \leq (x \vee y) \wedge z$.	(Modular *in*equality)

A *Boolean algebra* is a lattice with $\perp$ and T which is modular, distributive and complemented.

If $x \leq z$, then $x \vee (y \wedge z) = (x \vee y) \wedge z$	(Modularity)
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$,	(Distributivity)
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	
Let x' denote the *complement* of x.	(Complemented)
$x \wedge x' = \perp$, $x \vee x' = T$, $x'' = x$	
$(x \wedge y)' = x' \vee y'$, $(x \vee y)' = x' \wedge y'$	(DeMorgan)

Appendix II. Simple implementation of `subtypep` for a trivial type system

The following code is purely pedagogical; no production system would implement `subtypep` exactly this way. In particular, we apologize for the use of side-effects. However, for "anonymous" types created by `member`, one must either register them for the current call to `subtypep`, or register them once and for all. Depending upon the situation, either version could be more efficient: the temporary registration policy would be better if the type is only used once, while the permanent registration policy would be better if the type is used many times—e.g., if `subtypep` were called with the anonymous type within a loop.

```
(defconstant *standard-type-specifiers*
  '(FIXNUM BIGNUM FLOAT RATIO COMPLEX INTEGER RATIONAL CHARACTER
    NULL SYMBOL KEYWORD LIST etc.)
  "List of all built-in type names")

(defconstant *representatives*
  '(0 43574358734543 1.0 3/4 #C(1 1) #\S nil t :foo
    #(1 2) "Foo" #*10110 #2A((1 2) (3 4)) #.*standard-input*)
  "List of representatives for all equivalence classes")

(defun rep-bit (elt)
  "Return a bit-vector with bit set for this element"
  (let ((pos (position elt *representatives*)))
    (if pos (ash 1 pos)
        (progn (nconc *representatives* `(,elt)) (rep-bit elt)))))

(defun register (elt)
  "Register this element with all known types; return rep-bit"
  (dolist (typ *standard-type-specifiers*)
    (if (and (typep elt typ)
             (zerop (logand (rep-bit elt) (eval typ))))
        (incf (symbol-value typ) (rep-bit elt))))
  (rep-bit elt))

(defun elements (&rest elts)
  "Register elements and return OR of element bits"
  (reduce #'logior (mapcar #'register elts)))

(dolist (typ *standard-type-specifiers*) (set typ 0))

(apply #'elements *representatives*) ; Register all representatives

(defun subtypep (x y)
  "Predicate for subtype testing; map into bit-vector algebra"
  (let ((exp (sublis '((and . logand) (or . logior)
                       (not . lognot) (member . elements))
                     `(and ,x (not ,y)))))
  (eval exp) ; Dress rehersal to make sure elements are registered.
  (values (zerop (eval exp)) t))
```