DECL

Introduction

The Decl Library package is contained on the file DECL.DCOM. The Decl package requires the LambdaTran package. LAMBDATRAN.DCOM will automatically be loaded with Decl if it is not already present.

The Decl package extends Interlisp to allow the user to declare the types of variables and expressions appearing in functions. It provides a convenient way of constraining the behavior of programs when the generality and flexibility of ordinary Interlisp is either unnecessary, confusing, or inefficient.

Decl provides a simple language for declarations, and augments the interpreter and the compiler to guarantee that these declarations are always satisfied. The declarations make programs more readable by indicating the type, and therefore something about the intended usage, of variables and expressions in the code. They facilitate debugging by localizing errors that manifest themselves as type incompatibilities. Finally, the declaration information is available for other purposes: compiler macros can consult the declarations to produce more efficient code; coercions for arguments at user interfaces can be automatically generated; and the declarations will be noticed by the Masterscope function analyzer.

The declarations interpreted by the Decl package are in terms of a set of declaration types called *decltypes*, each of which specifies a set of acceptable values and also (optionally) other type-specific behavior. The Decl package provides a set of facilities for defining decltypes and their relations to each other, including type-valued expressions and a comprehensive treatment of union types.

The following description of the Decl package is divided into three parts. First, the syntactic extensions that permit the concise attachment of declarations to program elements are discussed. Second, the mechanisms by which new decltypes can be defined and manipulated are covered. Finally, some additional capabilities based on the availability of declarations are outlined.

Using Declarations in Programs

Declarations may be attached to the values of arbitrary expressions and to LAMBDA and PROG variables throughout (or for part of) their lexical scope. The declarations are attached using constructs that resemble the ordinary Interlisp LAMBDA, PROG, and PROGN, but which also permit the expression of declarations. The following examples illustrate the use of declarations in programs.

Consider the following definition for the factorial function (FACT *N*):

```
[LAMBDA (N)
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

Obviously, this function presupposes that *N* is a number, and the run-time checks in ITIMES and SUB1 will cause an error if this is not so. For instance, (FACT T) will cause an error and print the message NON-NUMERIC ARG T. By defining FACT as a DLAMBDA, the Decl package analog of LAMBDA, this presupposition can be stated directly in the code:

```
[DLAMBDA ((N NUMBERP))
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

With this definition, (FACT T) will result in a NON-NUMERIC ARG T error when the body of the code is executed. Instead, the NUMBERP declaration will be checked when the function is first entered, and a *declaration fault* will occur. Thus, the message that the user will see will not dwell on the offending value T, but instead give a symbolic indication of what variable and declaration were violated, as follows:

```
DECLARATION NOT SATISFIED
((N NUMBERP) BROKEN):
```

The user is left in a break from which the values of variables, e.g., *N*, can be examined to determine what the problem is.

The function FACT also makes other presuppositions concerning its argument, *N*. For example, FACT will go into an infinite recursive loop if *N* is a number less than zero. Although the user could program an explicit check for this unexpected situation, such coding is tedious and tends to obscure the underlying algorithm. Instead, the requirement that *N* not be negative can be succinctly stated by declaring it to be a subtype of NUMBERP that is restricted to non-negative numbers. This can be done by adding a SATISFIES clause to *N*'s type specification:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N])
   (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

The predicate in the SATISFIES clause will be evaluated after *N* is bound and found to satisfy NUMBERP, but before the function body is executed. In the event of a declaration fault, the SATISFIES condition will be included in the error message. For example, (FACT -1) would result in:

```
DECLARATION NOT SATISFIED
((N  NUMBERP  (SATISFIES  (NOT  (MINUSP  N)))
BROKEN):
```

The DLAMBDA construct also permits the type of the value that is returned by the function to be declared by means of the pseudo-variable RETURNS. For example, the following definition specifies that FACT is to return a positive integer:

```
[DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
  (COND
      ((EQ N 0) 1)
       (T (ITIMES N (FACT (SUB1 N]
```

After the function body is evaluated, its value is bound to the variable VALUE and the RETURNS declaration is checked. A declaration fault will occur if the value is not satisfactory. This prevents a bad value from propagating to the caller of FACT, perhaps causing an error far away from the source of the difficulty.

Declaring a variable causes its value to be checked not only when it is first bound, but also whenever that variable is reset by SETQ within the DLAMBDA. In other words, the type-checking machinery will not allow a declared variable to take on an improper value. An iterative version of the factorial function illustrates this feature in the context of a DPROG, the analog of PROG:

```
(DLAMBDA ([N NUMBERP (SATISFIES (NOT (MINUSP N]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
[DPROG ([TEMP 1 FIXP (SATISFIES (IGREATERP TEMP
0]
   [RETURNS FIXP (SATISFIES (IGREATERP VALUE 0])
      LP  (COND ((EQ N 0) (RETURN TEMP)))
          (SETQ TEMP (ITIMES N TEMP))
          (SETQ N (SUB1 N))
          (GO LP]
```

DPROG declarations are much like DLAMBDA declarations, except that they also allow an initial value for the variable to be specified. In the above example, TEMP is declared to be a positive integer throughout the computation and N is declared to be non-negative. Thus, a bug which caused an incorrect value to be assigned by one of the SETQ expressions would cause a declaration failure. Note that the RETURNS declaration for a DPROG is also useful in detecting the common bug of omitting an explicit RETURN.

DLAMBDAs

The Decl package version of a LAMBDA expression is an expression beginning with the atom DLAMBDA. Such an expression is a function object that may be used in any context where a LAMBDA expression may be used. It resembles a LAMBDA expression except that it permits declaration expressions in its argument list, as illustrated in the examples given earlier. Each element of the argument list of a DLAMBDA may be a literal atom (as in a conventional LAMBDA) or a list of the form (*NAME TYPE .EXTRAS*). Strictly, this would require a declaration with a SATISFIES clause to take the form (N (NUMBERP (SATISFIES --)) --). However, due to the frequency with which this construction is used, it may be written without the inner set of parentheses, e.g., (N NUMBERP (SATISFIES --) --).

NAME fulfills the standard function of a parameter, i.e., providing a name to which the value of the corresponding argument will be bound.

TYPE is either a Decl package type name or type expression. When the DLAMBDA is entered, its arguments will be evaluated and bound to the corresponding argument names, and then, after all the argument names have been bound, the declarations will be checked. The type checking is delayed so that SATISFIES predicates can include references to other variables bound by the same DLAMBDA. For example, one might wish to define a function whose two arguments are not only both required to be of some given type, but are also required to satisfy some relationship (e.g., that one is less than the other).

EXTRAS allows some additional properties to be attached to a variable. One such property is the accessibility of *NAME* outside the current lexical scope. Accessibility specifications include the atoms LOCAL or SPECIAL, which indicate that this variable is to be compiled so that it is either a LOCALVAR or a SPECVAR, respectively. This is illustrated by the following example:

```
[DLAMBDA ((A LISTP SPECIAL)
          (B FIXP LOCAL))
          ...]
```

A more informative equivalent to the SPECIAL key word is the USEDIN form, the tail of which can be a list of the other functions that are expected to have access to the variable.[1]

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE))
      (B FIXP LOCAL))
      ...]
```

EXTRAS may also include a comment in standard format, so that descriptive information may be given where a variable is bound:

```
[DLAMBDA ((A LISTP (USEDIN FOO FIE)   (* This is
an important variable))
          (B FIXP LOCAL))
          ...]
```

As mentioned earlier, the value returned by a DLAMBDA can also be declared, by means of the pseudo-variable RETURNS. The RETURNS declaration is just like other DLAMBDA declarations, except (1) in any SATISFIES predicate, the value of the function is referred to by the distinguished name VALUE; and (2) it makes no sense to declare the return value to be LOCAL or SPECIAL.

DPROG

Just as DLAMBDA resembles LAMBDA, DPROG is analogous to PROG. As for an ordinary PROG, a variable binding may be specified as an atom or a list including an initial value form. However, a DPROG binding also allows *TYPE* and *EXTRAS* information to appear following the initial value form. The format for these augmented variable bindings is (*NAME INITIALVALUE TYPE .EXTRAS*).

The only difference between a DPROG binding and a DLAMBDA binding is that the second position is interpreted as the initial value for the variable. Note that if the user wishes to supply a type declaration for a variable, an initial value *must* be specified. The same rules apply for the interpretation of the type information for DPROGs as for DLAMBDAs, and the same set of optional *EXTRA*s can be used. DPROGs may also declare the type of the value they return, by specifying the pseudo-variable RETURNS.

Just as for a DLAMBDA, type tests in a DPROG are not asserted until after all the variables have been bound, thus permitting predicates to refer to other variables being bound by this DPROG. If NIL appears as the initial value for a binding (i.e., the atom NIL actually appears in the code, not simply an expression that evaluates to NIL) the initial type test will be suppressed, but subsequent type tests, e.g., following a SETQ, will still be performed.

A common construct in Lisp is to bind and initialize a PROG variable to the value of a complicated expression in order to avoid recomputing it, and then to use this value in initializing other PROG variables, e.g.

```
[PROG ((A EXPRESSION))
      (RETURN (PROG ((B... ( A...))
                     (C... ( A... )))
                    ...]
```

The ugliness of such constructions in conventional Lisp often tempts the programmer to loosen the scoping relationships of the variables by binding them all at a single level and using SETQ's in the body of the PROG to establish the initial values for variables that depend on the initial values of other variables, e.g.,

```
[PROG ((A  EXPRESSION) B C)
      (SETQ B (...A... ))
      (SETQ C ( ...A... ))
      ...]
```

In the Decl package environment, this procedure undermines the protection offered by the type mechanism by encouraging the use of uninitialized variables. Therefore, the DPROG offers a syntactic form to encourage more virtuous initialization of its variables. A DPROG variable list may be segmented by occurrences of the special atom THEN, which causes the binding of its variables in stages, so that the bindings made in earlier stages can be used in later ones, e.g.,

```
[DPROG ((A (LENGTH FOO) FIXP LOCAL)
        THEN (B (SQRT A) FLOATP)
        THEN (C (CONS A B) LISTP))
        ...]
```

Each stage is carried out as a conventional set of DPROG bindings (i.e., simultaneously, followed by the appropriate type testing). This layering of the bindings permits one to gradually descend into a inner scope, binding the local names in a very structured and clean fashion, with initial values type-checked as soon as possible.

Declarations in Iterative Statements

The CLISP iterative statement provides a very useful facility for specifying a variety of PROGs that follow certain widely used formats. The Decl package allows declarations to be made for the scope of an iterative statement via the DECLARE CLISP (I.S. operator). DECLARE can appear as an operator anywhere in an iterative statement, followed by a list of declarations, for example:

(for J from 1 to 10 declare (J FIXP) do...

Note that DECLARE declarations do not create bindings, but merely provide declarations for existing bindings. For this reason, an initial value cannot be specified and the form of the declaration is the same as that of DLAMBDAs, namely create (*NAME TYPE . EXTRAS*).

Note that variables bound *outside* of the scope of the iterative statement, i.e., a variable used freely in the I.S., can also be declared using this construction. Such a declaration will only be in effect for the scope of the iterative statement.

Declaring a Variable for a Restricted Lexical Scope

The Decl package also permits declaring the type of a variable over some restricted portion of its existence. For example, suppose the variable X is either a fixed or floating number, and a program branches to treat the two cases separately. On one path X is known to be fixed, whereas on the other it is known to be floating. The Decl package DPROGN construct can be used in such cases to state the type of the variable along each path. DPROGN is exactly like PROGN, except that the second element of the form is interpreted as a list of DLAMBDA format declarations. These declarations are added to any existing declarations in the containing scope, and the composite declaration (created using the ALLOF type expression), is considered to hold throughout the lexical scope created by the DPROGN. Thus, our example becomes:

```
(if (FIXP X)
then (DPROGN ((X FIXP))...else (DPROGN ((X
FLOATP)) ...))
```

Like DPROG and DLAMBDA, the value of a DPROGN may also be declared, using the pseudo-variable RETURNS.

DPROGN may be used not only to restrict the declarations of local variables, but also to declare variables that are being used freely. For example, if the variable A is used freely inside a function but is known to be FIXP, this fact could be noted by enclosing the body of the function in (DPROGN ((A FIXP FREE)) *BODY*). Instead of FREE, the more specific construction (BOUNDIN *FUNCTION1 FUNCTION 2*. . .) can be used. This not only states that the variable is used freely but also gives the names of the functions that might have provided this binding.[2]

Since the DPROGN form introduces another level of parenthesization, which results in the enclosed forms being prettyprinted indented, the Decl package also permits such declarations to be attached to their enclosing DLAMBDA or DPROG scopes by placing a DEC expression, e.g., (DECL (A FIXP (BOUNDIN FUM)), before the first executable form in that scope. Like DPROGN's, DECL declarations use DLAMBDA format.

Declaring the Values of Expressions

The Decl package allows the value of an arbitrary form to be declared with the Decl construct THE. A THE expression is of the form (THE *TYPE . FORMS*), e.g., (THE FIXP (FOO X)). FORMS are evaluated in order, and the value of the *last* one is checked to see if it satisfies *TYPE*, a type name or type expression. If so, its value is returned, otherwise a declaration fault occurs.

Assertions

The Decl package also allows for checking that an arbitrary predicate holds at a particular point in a program's execution, e.g., a condition that must hold at function entry but not throughout its execution. Such predicates can be checked using an expression of the form (ASSERT *FORM1 FORM2*), in which each *FORM1* is either a list (which will be evaluated) or a variable (whose declaration will be checked). Unless all elements of the ASSERT form are satisfied, a declaration fault will take place.

ASSERTing a variable provides a convenient way of verifying that the value of the variable has not been improperly changed by a lower function. Although a similar effect could be achieved for predicates by explicit checks of the form (OR *PREDICATE* (SHOULDNT)), ASSERT also provides the ability both to check that a variable's declaration is currently satisfied and to remove its checks at compile time without source code modification (see COMPILEIGNOREDECL).

Using Type Expressions as Predicates

The Decl package extends the Record package TYPE? construct so that it accepts decltypes, as well as record names, e.g., (TYPE? (FIXP (SATISFIES (ILESSP VALUE 0))) EXPR). Thus, a TYPE? expression is exactly the same as a THE expression except that, rather than causing a declaration fault, TYPE? is a predicate that determines whether or not the value satisfies the given type.

Enforcement

The Decl package is a "soft" typing system—that is, the data objects themselves are not inherently typed. Consequently, declarations can only be enforced within the lexical scope in which the declaration takes place, and then only in certain contexts. In general, changes to a variable's value such as those resulting from side effects to embedded structure (e.g., RPLACA, SETN, etc.) or free variable references from outside the scope of the declaration cannot be, and therefore are not, enforced.

Declarations *are* enforced, i.e., checked, in three different situations: when a declared variable is bound to some value or rebound with SETQ or SETQQ, when a declared expression is evaluated, and when an ASSERT expression is evaluated. In a binding context, the type check takes place *after* the binding, including any user-defined behavior specified by the type's binding function. Any failure of the declarations causes a break

to occur and an informative message to be printed. In that break, the name to which the declaration is attached (or *VALUE* if no name is available) will be bound to the offending value. Thus, in the FACT T example above, N would be bound to T. The problem can be repaired either by returning an acceptable value from the break via the RETURN command, or by assigning an acceptable value to the offending name and returning from the break via an OK or GO command. The unsatisfied declaration will be reasserted when the computation is continued, so an unacceptable value will be detected.[3]

The automatic enforcement of type declarations is a very flexible and powerful aid to program development. It does, however, exact a considerable run-time cost because of all the checking involved. Factors of two to ten in running speed are not uncommon, especially where low-level, frequently used functions employ type declarations. As a result, it is usually desirable to remove the declaration enforcement code when the system is believed to be bug-free and performance becomes more central. This can be done with the variable COMPILEIGNOREDECL.

COMPILEIGNOREDECL [Variable]

Setting the value of the variable COMPILEIGNOREDECL to T (initially NIL) instructs the compiler not to insert declaration enforcement tests in the compiled code. More selective removal can be achieved by setting COMPILEIGNOREDECL to a list of function names. Any function whose name is found on this list is compiled without declaration enforcement.

IGNOREDECL. *VAL* [File Com]

Declaration enforcement may be suppressed selectively by a file using the IGNOREDECL file package command. If this appears in a file's file commands, it redefines the value of COMPILEIGNOREDECL to VAL for the compilation of this file only.

Note: The period in the IGNOREDECL file package command is significant. To set COMPILEIGNOREDECL to T, use (IGNOREDECL . T), not (IGNOREDECL T).

Decltypes

A Decl package type, or decltype, specifies a subset of data values to which values of this type are restricted. For example, a "positive number" type might be defined to include only those values that are numbers and greater than zero. A type may also specify how certain operations, such as assignment or binding (see BINDFN), are to be performed on variables declared to be of this type.

The inclusion relations among the sets of values that satisfy the different types define a natural partial ordering on types, bound by the universal type ANY (which all values satisfy) and the empty type NONE (which no value satisfies). Each type has one or more *supertypes* (each type has at least ANY as a supertype) and one or more subtypes (each type has at least NONE as a subtype). This structure is important to the user of Decl as it provides the framework in which new types are defined. Typically, much of the definition of a new type is defaulted, rather than specified explicitly. The definition will be completed by inheriting attributes which are shared by all its immediate supertypes.

An initial set of decltypes that defines the Interlisp built-in data types and a few other commonly used types is provided. Thereafter, new decltypes are created in terms of existing ones using the type expressions described below. For conciseness, such new types can be associated with literal atoms using the function DECLTYPE.

Predefined Types

Some commonly used types, such as the Interlisp built-in data types, are already defined when the Decl package is loaded. These types, indented to show subtype-supertype relations, are:[4]

```
ANY
  ATOM              LST
     ARRAYP       STRINGP     FUNCTION     STACKP
    LITATOM          ALIST     HARRAYP
      NIL            LISTP        READTABLEP
    NUMBERP
      FIXP
        LARGEP
        SMALLP
      FLOATP

                                          NONE
```

Note that the definition of LST causes NIL to have multiple supertypes, i.e., LITATOM and LST, reflecting the duality of NIL as an atom and a (degenerate) list.

In addition, declarations made using the Record package also define types that are attached as subtypes to an appropriate existing type (e.g., a TYPERECORD declaration defines a subtype of LISTP, a DATATYPE declaration a subtype of ANY, etc.) and may be used directly in declaration contexts.

Type Expressions

Type expressions provide convenient ways for defining new types in terms of modifications to, or compositions of one or more existing types.

(MEMQ *VALUE1...VALUE N*) [Type Expression]

Specifies a type whose values can be any one of the fixed set of elements VALUE 1. . .VALUE N. For example, the status of a device might be represented by a datum restricted to the values BUSY and FREE. Such a "device status" type could be defined via (MEMQ BUSY FREE). The new type will be a subtype of the narrowest type that all of the alternatives satisfy (e.g., the "device status" type would be a subtype of LITATOM). The membership test uses EQ if this supertype is a LITATOM; EQUAL otherwise. Thus, lists, floating point numbers, etc., can be included in the set of alternatives.

(ONEOF *TYPE 1...TYPE N*) [Type Expression]

Specifies a type that is the union of two or more other types. For example, the notion of a possibly degenerate list is something that is either LISTP or NIL. Such a type can be (and the built-in type LST in fact is) defined simply as (ONEOF NIL LISTP). A union data type becomes a supertype of all of the alternative types specified in the ONEOF expression, and a subtype of their lowest common supertype. The type properties of a union type are taken from its alternative types if they all agree, otherwise from the supertype.

(ALLOF *TYPE 1...TYPE N*) [Type Expression]

Specifies a type that is the intersection of two or more other types. For example, a variable may be required to satisfy both FIXP and also some type that is defined as (NUMBERP (SATISFIES *PREDICATE*)). The latter type will admit numbers that are not FIXP, i.e., floating point numbers; the former does not include *PREDICATE*. Both restrictions can be obtained by using the type (ALLOF (NUMBERP (SATISFIES *PREDICATE*)) FIXP).[5]

(OF *AGGREGATE* OF *ELEMENT*) [Type Expression]

Specifies *DECLaggregate,* a type that is an aggregate of values of some other type (e.g., list of numbers, array of strings, etc.). *AGGREGATE* must be a type that provides an EVERYFN property. The EVERYFN is used to apply an arbitrary function to each of the elements of a datum of the aggregate type, and check whether the result is non-NIL for each element. *ELEMENT* may be any type expression. For example, the type "list of either strings or atoms" can be defined as (LISTP OF (ONEOF STRINGP ATOM)). The type test for the new type will consist of applying the type test for *ELEMENT* to each element of the aggregate type using the EVERYFN property. The new type will be a subtype of its aggregate type.[6]

(SATISFIES *TYPE* (SATISFIES *FORM 1. . .FORM N*)) [Type Expression]

Specifies a type whose values are a subset of the values of an existing type. The type test for the new type will first check that the base type is satisfied, i.e., that the object is a member of TYPE, and then evaluate *FORM 1. . .FORM N*. If each form returns a non-NIL value, the type is satisfied.

The value that is being tested may be referred to in *FORM 1. . . FORM N* by either (a) the variable name if the type expression appears in a binding context such as DLAMBDA or DPROG, (b) the distinguished atom ELT for a SATISFIES clause on the elements of an aggregate type, or (c) the distinguished atom VALUE, when the type expression is used in a context where no name is available (e.g., a RETURNS declaration). For example, one might declare the program variable A to be a negative integer via (FIXP (SATISFIES (MINUSP A))) or declare the value of a DLAMBDA to be of type ((ONEOF FIXP FLOATP) (SATISFIES (GREATERP VALUE 25))).

Note that more than one SATISFIES clause may appear in a single type expression attached to different alternatives in a ONEOF type expression, or attached to both the elements and the overall structure of an aggregate. For example,

```
[LISTP OF [FIXP (SATISFIES (ILEQ ELT (CAR VALUE]
          (SATISFIES (ILESSP (LENGTH VALUE) 7]
```

specifies a list of less than seven integers each of which is no greater than the first element of the list.

(SHARED *TYPE*) [Type Expression]

Specifies DECLshared, a subtype of *TYPE*, with default binding behavior, i.e., the binding function (see BINDFN), if any, will be suppressed.[7] For example, if the type FLOATP were redefined so that DLAMBDA and DPROG bindings of variables that were declared to be FLOATP copied their initial values (e.g., to allow SETNs to be free of side effects), then variables declared (SHARED FLOATP) would be initialized in the normal fashion, without copying their initial values.

Named Types

Although type expressions can be used in any declaration context, it is often desirable to save the definition of a new type if it is to be used frequently, or if a more complex specification of its behavior is to be given than is convenient i. an expression. The ability to define a named type is provided by the function DECLTYPE.

(DECLTYPE *TYPENAME TYPE PROP1 VAL1 PROPN VALN*) [Function]

NLambda, nospread function. *TYPENAME* is a literal atom, *TYPE* is either the name of an existing type or a type expression, and *PROP 1, VAL 1...PROP N,VAL N* is a specification (in property list format) of other attributes of the type. DECLTYPE derives a type from *TYPE*, associates it with *TYPENAME*, and then defines any properties specified with the values given.

The following properties are interpreted by the Decl package.[8] Each of these properties can have as its value either a function name or a LAMBDA expression.

TESTFN [Property]

will be used by the Decl package to test whether a given value satisfies this type. The type is considered satisfied if *FN* applied to the item is non-NIL. For example, one might define the type INTEGER with TESTFN FIXP.[9]

EVERYFN [Property]

EVERYFN specifies a mapping function that can apply a functional argument to each "element" of an instance of this type, and which will return NIL unless the result of every such application was non-NIL. *FN* must be a function of two arguments: the aggregate and the function to be applied. For example, the EVERYFN for the built-in type LISTP is EVERY. The Decl package uses the EVERYFN property of the aggregate type to construct a type test for aggregate type expressions. In fact, it is the presence of an EVERYFN property that allows a type to be used as an aggregate type.[10]

BINDFN [Property]

BINDFN is used to compute from the initial value supplied for a DLAMBDA or DPROG variable of this type, the value to which the variable will actually be initialized. *FN* must be a function of one argument that will be applied to the initial value, and which should produce another value which is to be used to make the binding.[11] For example, a BINDFN could be used to bind variables of some type so that new bindings are copies of the initial value. Thus, if FLOATP were given the BINDFN FPLUS, any variable declared FLOATP would be initialized with a new floating box, rather than sharing with that of the original initial value.[12]

SETFN [Property]

is used for performing a SETQ or SETQQ of variables of this type. *FN* is a function of two arguments, the name of the variable and its new value. A SETFN is typically used to avoid the allocation of storage for intermediate results. Note that the SETFN is not the mechanism for the enforcement of type compatibility, which is checked *after* the assignment has taken place. Also note that not all functions that can change values are affected: in particular, SET and SETN are not.

Manipulating Named Types

DECLTYPES is a file package type. Thus all of the operations relating to file package types, e.g., GETDEF, PUTDEF, EDITDEF, DELDEF, SHOWDEF, etc., can be performed on decltypes.[13]

The file package command, DECLTYPES , is provided to dump named decltypes symbolically. They will be written as a series of DECLTYPE forms that will specify only those fields that differ from the corresponding field of their supertype(s). If the type depends on any unnamed types, those types will be dumped (as a compound type expression), continuing up the supertype chain until a named type is found. Care should be exercised to ensure that enough of the named type context is dumped to allow the type definition to remain meaningful.

The functions GETDECLTYPEPROP and SETDECLTYPEPROP, defined analogously to the property list functions for atoms, allow the manipulation of the properties of named types. Setting a property to NIL with SETDECLTYPEPROP removes it from the type.

Relations Between Types

The notion of equivalence of two types is not well defined. However, type equivalence is rarely of interest. What is of interest is type *inclusion*, i.e., whether one type is a supertype or subtype of another. The predicate COVERS can be used to determine whether the values of one type include those of another.

(COVERS *HI LO*) [Function]

COVERS is T if *HI* can be found on some (possibly empty) supertype chain of *LO*; else NIL. Thus, (COVERS 'FIXP (DECLOF 4)) = T, even though the DECLTYPE of four is SMALLP, not FIXP. The extremal cases are the obvious identities:

(COVERS 'ANY ANYTYPE) = (COVERS ANYTYPE 'NONE) = (COVERS X) for any type X = T.

COVERS allows declaration-based transformations of a form that depend on elements of the form being of a certain type to express their applicability conditions in terms of the weakest type to which they apply, without explicit concern for other types that may be subtypes of it. For example, if a particular transformation is to be applied whenever an element is of type NUMBERP, the program that applies that transformation does not have to check whether the element is of type SMALLP, LARGEP, FIXP, FLOATP, etc., but can simply ask whether NUMBERP COVERS the type of that element.

The elementary relations among the types, out of which arbitrary traversals of the type space can be constructed, are made available via:

(SUBTYPE *TYPE*) [Function]

Returns the list of types that are *immediate* subtypes of *TYPE*.

(SUPERTYPES *TYPE*) [Function]

Returns the list of types that are *immediate* supertypes of *TYPE*.

The Declaration Data Base

One of the primary uses of type declarations is to provide information that other systems can use to interpret or optimize code. For example, one might choose to write all arithmetic operations in terms of general functions like PLUS and TIMES and then use variable declarations to substitute more efficient, special-purpose code at compile time based on the types of the operands. To this end, a data base of declarations is made available by the Decl package to support these operations.

(DECLOF *FORM*) [Function]

Returns the type of *FORM* in the current declaration context. If *FORM* is an atom, DECLOF will look up that atom directly in its data base of current declarations. Otherwise, DECLOF will look on the property list of (CAR *FORM*) for a DECLOF property, as described below. If there is no DECLOF property, DECLOF will check if (CAR *FORM*) is one of a large set of functions of known result type (e.g., the arithmetic functions). Failing that, if (CAR *FORM*) has a MACRO property, DECLOF will apply itself to the result of expanding (with EXPANDMACRO), the macro definition. Finally, if *FORM* is a Lisp program element that DECLOF "understands" (e.g., a COND, PROG, SELECTQ, etc.), DECLOF applies itself recursively to the part(s) of the contained form which will be returned as value.[14]

DECLOF [Property]

Allows the specification of the type of the values returned by a particular function. The value of the DECLOF property can be either a type, i.e., a type name or a type expression, or a list of the form (FUNCTION *FN*), where *FN* is a function object. *FN* will be applied (by DECLOF) to the form whose CAR has this DECLOF property on its property list. The value of this function application will then be considered to be the type of the form.

As an example of how declarations can be used to automatically generate more efficient code, consider an arithmetic package Declarations of numeric variables could be used to guide code generation to avoid the inefficiencies of Interlisp's handling of arithmetic values. Not only could the generic arithmetic functions be automatically specialized, as suggested above, but

by redefining the BINDFN and the SETFN properties for the types FLOATP and LARGEP to reuse storage in the appropriate contexts (i.e., when the new value can be determined to be of the appropriate type), tremendous economies could be realized by not allocating storage to intermediate results that must later be reclaimed by the garbage collector. The Decl package has been used as the basis for several such code optimizing systems.

Declarations and Masterscope

The Decl package notifies MASTERSCOPE about type declarations and defines a new MASTERSCOPE relation, TYPE, which depends on declarations. Thus, the user can ask questions such as "WHO USES MUMBLE AS A TYPE?," " DOES FOO USE FIXP AS A TYPE?," and so on.

End Notes

1. USEDIN is mainly for documentation purposes, since there is no way for such a restriction to be enforced.
2. Like USEDIN declarations, FREE and BOUNDIN declarations cannot be checked, and are for documentation purposes only.
3. With this exception, assignments to variables from within the break are not considered to be in the scope of the declarations that were in effect when the break took place and so are not checked.
4. LST is defined as either LISTP or NIL. i.e., a list or NIL. The name LST is used because the name LIST is treated specially by CLISP. A LIST is defined as either NIL or a list of elements each of which is of type LISTP.
5. When a value is tested, the component type tests are applied from left to right.
6. The built-in aggregate types are ARRAP, LISTP, LST, and STRINGP (and their subtypes).
7. As no predefined type has a binding function, this is of no concern until the user defines or redefines a type to have a binding function.
8. Actually, any property can be attached to a type, and will be available for use by user functions via the function GETDECLTYPEPROP.
9. Typically, the TESTFN for a type is derived from its type expression, rather than specified explicitly. The ability to

specify the TESTFN is provided for those cases where a predicate is available that is much more efficient than that which would be derived from the type expression. For example, the type SMALLP is defined to have the function SMALLP as its TESTFN, rather than (LAMBDA(DATUM) (AND(NUMBERP DATUM)(FIXP DATUM) (SMALLP DATUM))) as would be derived from the subtype structure.

10. Note that a type's EVERYFN is not used in type tests for that type, but only in type tests for types defined by OF expressions that used this type as the aggregate type. For example, EVERY is not used in defining whether some value satisfies the type LISTP. The Decl package never applies the EVERYFN of a type to a value without first verifying that the value satisfies that type.

11. For a PPROG binding, FN will be applied to no arguments if the initial value is lexically NIL.

12. The BINDFN, if any, associated with a type may be suppressed in a declaration context by creating a subtype with the type-expressing operator SHARED.

13. Deleting a named type could possibly invalidate other type definitions that have the named type as a subtype or supertype. Consequently, the deleted type is simply unnamed and left in the type space as long as it is needed.

14. "The current declaration context" is defined by the environment at the time that DECLOF is called. Code-reading systems, such as the compiler and the interpreter, keep track of the lexical scope within which they are currently operating, in particular, which declarations are in effect. Note that (currently) DECLOF does *not* have access to any global data base of declarations. For example, DECLOF does not have information available about the types of arguments of, or the value returned by, a particular function, unless it is currently "inside" that function. However, the DECLOF property can be used to inform DECLOF of the type of the value returned by a particular function.