Abstract data types, subtypes and data independence

F. Warren Burton† and Brian J. Lings‡

A generalised subtype definition facility is described. A subtype may be restricted in its values, its operations or both. Coercion from a subtype value to a parent type value is always possible. The reverse coercion may or may not be allowed, depending on the subtype. A subtype may or may not use the same form of representation as the parent type. Several examples showing the usefulness of the subtype facility are given. It is shown that, with a few additional facilities, subtypes may be useful in providing data independence in a general purpose language. Finally, it is shown that the concept of subtype is useful in understanding the relationships between certain other programming language constructs.

(Received March 1980)

1. Introduction

The concept of subtype has existed in programming languages in an ad hoc manner for many years. In FORTRAN (ANSI, 1966), INTEGER may be viewed as a subtype of REAL. Coercion from one type to the other is possible in an assignment statement. More recently, Pascal (Jensen and Wirth, 1974) has provided subrange types where a subrange type may be viewed as a subtype of the associated scalar type. Most recently, Ada (ACM, 1979) has introduced the keyword **subtype** and generalised the Pascal nesting of types.

However, a more general subtype mechanism than exists in any of these languages is required. In Ichbiah *et al.* (1979, p. 4–1), it is stated that "A now widely accepted view of types is that a type characterizes the set of values that objects of the type may assume, and the set of operations that may be performed on them." With this view, a subtype construct should provide more than simple value restrictions on restricted classes of types. It should be possible to define subtypes of a type where arbitrary restrictions may be placed on the values, operations or both. Subtypes of arbitrary user defined types should be supported. For example, QUEUE could be a subtype of LIST with restricted operations while a SORTED_LIST could be a subtype with restricted values (and perhaps operations as well). Certain operations, for example a diagnostic procedure which prints the contents of a LIST, should be able to process a LIST, a QUEUE or a SORTED_LIST.

Usually INTEGERS and REALS are represented differently. In like manner, it should be possible for user defined subtypes and types to have different representations. (We note that in Ada, INTEGER is not a subtype of REAL.) For example, if approximations to curves are represented by polygons, it is reasonable to have a subtype for circles using a centre–radius representation.

When a subtype is defined, it should always be possible to coerce a value of the subtype to a value of the parent type. (If A is a subtype of B we will call B the parent type of A.) For example, in Pascal it is possible to assign an INTEGER to a REAL variable or a subrange value to a variable having the associated scalar type. The definition of a subtype should specify whether the reverse coercion is allowed. For example, in Pascal the reverse coercion is allowed with subranges, but a REAL may not be assigned to an INTEGER variable. For uniformity, coercions between actual and formal parameters should also be supported. (The distinction between input, output and input–output parameters in languages such as Ada is most helpful in this context.)

2. Ada extension

For purposes of illustration we shall present the proposed language features as an extension to Ada as we understand the language from ACM (1979). We will assume that the reader is at least vaguely familiar with Ada.

The concept of subtype exists in Ada in a limited form. To fully support subtypes it is necessary to extend Ada to provide a mechanism for specifying an arbitrary constraint for the values of a subtype. In addition, it must be possible to specify an alternative representation for a subtype and to define suitable conversion operations.

The syntax of a subtype declaration may be modified as shown in **Fig. 1.** A **by** clause has been added to the definition of a subtype declaration and a functional constraint has been added to the definition of constraint. The type mark following the **by** specifies the representation of the subtype. The function specified in a functional constraint must be a Boolean function with a single parameter whose value is of the parent type.

subtype_declaration ::=
 subtype *identifier* **is** *type_mark* [*constraint*] [**by** *type_mark*]
constraint ::=
 range_constraint | *accuracy_constraint*
 | *index_constraint* | *discriminant_constraint*
 | *functional_constraint*
functional_constraint ::= **which is** *function_name*

Fig. 1 Proposed extension to the syntax of Ada

Fig. 2 shows how a subtype EVEN of INTEGER may be defined. Since the definition of EVEN does not contain a **by** clause, conversion from the subtype to the parent type and from the parent type to the subtype are defined and trivial.

An alternative definition of EVEN, where INTEGER and EVEN are represented in different ways, is given in **Fig. 3.** Since in the latter example INTEGER and EVEN are to be represented in different ways, conversion operations must be defined by the user. If I is an INTEGER variable and E is an EVEN variable then the assignment $E := I$ will cause I to be tested by DIVISIBLE_BY_2. If the test fails then the exception CONVERSION_ERROR is raised, otherwise function "EVEN" performs the conversion and the result is assigned to E. We note that subtype EVEN of INTEGER could have been represented "by REAL" if we had wanted to make a more complicated example.

†School of Computing Studies, University of East Anglia, Norwich, NR4 7TJ, UK.
‡Department of Computer Science, University of Queensland, Brisbane, Australia 4067.

CCC-0010-4620/81/0024-0308 $02.00

```
function DIVISIBLE_BY_2 (I:INTEGER) return BOOLEAN is
  pragma INLINE;
 begin
 return I mod 2 = 0;
 end;
subtype EVEN is INTEGER which is DIVISIBLE_BY_2;
```

Fig. 2 Definition of subtype EVEN

```
subtype EVEN is INTEGER which is DIVISIBLE_BY_2 by INTEGER;
function "EVEN" (I:INTEGER) return EVEN is
  pragma INLINE;
 begin
  return I/2;
 end;
function "INTEGER" (E:EVEN) return INTEGER is
  pragma INLINE;
 begin
  return E*2;
 end;
```

Fig. 3 An alternative definition of EVEN

It must always be possible to coerce a subtype value to a parent type value. Therefore, the declaration of the "INTEGER" operator was mandatory. Provision of an operator to coerce from a parent type to a subtype is optional. If "EVEN" had not been declared then it would not have been possible to coerce an INTEGER into an EVEN. (By analogy, in Pascal it is not possible to coerce a REAL into an INTEGER.)

The pragma SUPPRESS may be used to turn off error checking where it is not required. Otherwise error checking is performed whenever a new value is assigned to a variable with a functional constraint unless the value is of the same subtype. Parameter passing is viewed as equivalent to assignment.

Care must be taken when using functional constraints with linked structures since it may be possible to access and modify substructures without assigning a new value to the constrained variable. This problem may be overcome by using encapsulation, in the form of packages, to prevent back door access to data.

We note that Ada could be further extended to permit functional constraints for types as well as subtypes, and to support supertypes as well as subtypes. For example, COMPLEX could be defined as the parent type of REAL.

3. Examples

Subtypes may be used to advantage in a number of ways. These include error checking, as illustrated by the example in the previous section. The reader will have no trouble imagining numerous other examples along these lines. In effect, a functional constraint is an assertion associated with all data structures of a particular type rather than with a point in the text of a program.

Subtypes may also be used to permit certain operations to be used for a collection of similar types in a controlled manner. For example suppose STACK, QUEUE, SORTED_LIST and READ_APPEND_LIST are all subtypes of LIST. Suppose that in every case coercion from LIST to the subtype is prohibited. Operations which modify LISTs (by having **out** or **in out** mode parameters of type LIST) can not be applied to any of the subtypes. However, safe operations which do not modify LISTs (where parameters of type LIST are of mode **in**) may be applied to the subtypes of LIST. In this way a single diagnostic procedure which prints the contents of a LIST may be used for any LIST subtype. If the subtypes are defined as illustrated in **Fig. 4** then the coercion from the subtype to LIST is not likely to involve any cost at all. (Since this situation

```
subtype STACK is LIST by LIST;
function "LIST" (S:STACK) return LIST is
  pragma INLINE;
 begin
  return S;
 end;
```

Fig. 4 Definition of subtype STACK

is fairly common, perhaps a special syntactic form such as

strict subtype STACK **is** LIST;

may be warranted.)

In a large program or system of programs it may be necessary to have several different ways of representing objects which logically are regarded as being of the same type. For example, there are a number of ways to represent sets. The best method of representation depends on the expected size of a set, the size of the universal set, the operations which are to be applied to sets and other factors. Any one method of representation will cause serious inefficiencies if used in the wrong situation. Therefore, a program may have to use several different methods for representing sets. It may be necessary on occasions for sets represented in different ways to interact, so distinct types for different representations is not adequate. Similarly, points in a plane may be represented in either polar or rectangular form. Another example along these lines will be considered in the next section.

Suppose we wish to use both polar points (for frequent complex exponentiation) and rectangular points (for frequent addition) in a program. We could declare either to be a subtype of the other. (Perhaps the keyword **cotype** could be introduced, where coercion both ways must be possible between **cotypes.**) In this way a variable could be declared in light of its expected usage, while the possibility of coercion would allow the programmer to use the variable freely. A simpler example can be seen in the area of unit conversion. Inches could be defined as

subtype INCHES **is** FEET **by** REAL;

where conversion between feet and inches would involve multiplication or division by 12. To date, representational alternatives in programming languages have been restricted to a few special cases, such as packed and unpacked structures, binary and character representations for numbers and structures of reduced size for records with a fixed variant part.

4. Data independence

Data independence (Date, 1975) has long been advocated in the context of data base systems. Data independence is the separation of the logical view and physical representation of data. For example, in a relational data base system a user thinks and programs in terms of abstract relations. In practice relations may be represented in any of a number of ways depending on how they are used. In a general purpose programming language it is desirable to be able to support data independence for a variety of logical data types.

The use of subtypes as discussed in the previous section is not really sufficient to support a reasonable level of data independence. The lack of a collective name for a family of types discourages a programmer from thinking in terms of a single logical type. In addition, problems arise when an operation is defined for more than one type (subtype) in the family. For example, if multiplication is defined for both POLAR and RECTANGULAR types, then a simple multiplication is ambiguous. In particular, if a programmer wishes to multiply a POLAR by a RECTANGULAR he must

declaration::=
  declaration_as_defined_in_Ada
 | *family_declaration* | *family_subprogram_declaration*
family_declaration::=
  family *identifier* **is** *family_definition*
family_definition::=
  type_mark | *type_mark* **only**
 | *type_mark* **excluding** (*type_mark* {, *type_mark*})
 | *type_mark* **including** (*family_definition* {, *family_definition*})
family_subprogram declaration::=
  as illustrated in Fig. 6

Fig. 5 Syntax extension for data independence through type families

specify which operator is to be used. The decision must be made each time multiplication is used rather than once when multiplication is defined.

The solution is to give a collective name to a family of types and to define operations for a family in such a way that the various possible parameter combinations are considered at the point of definition rather than at the points of invocation. This can be done if the syntax of Ada is extended as shown in **Fig. 5.**

The direct and indirect subtypes of a type together with the type itself form a tree. There is a unique path between any two nodes of the tree and so coercion from one subtype (or type) to another is unique if possible at all. A family is a connected subset of a type tree. A family defined by a type mark consists of an entire tree of types, including parent types, subtypes of parent types etc. (A type mark is a type or subtype name.) A singleton family is defined by a type mark followed by **only.** A restricted family can be defined by listing either the type marks to be excluded or the sub families to be included. If a type mark is excluded then all type marks joined to the tree via paths through the excluded type mark are also excluded. The family definition

A **including** (*B*)

is taken to mean

A **including** (*B* **excluding** (*A*))

since *A* need not be considered twice.

Since we are interested in the concept of data independence through type families, rather than a specific language proposal, we will not give the full syntax of a *family_subprogram_declaration.* With *family* firmly defined, an example should suffice.

Suppose POLAR is a subtype of RECTANGULAR. The definition of family POINT in **Fig. 6** includes both RECTANGULAR and POLAR and excludes any other subtypes either may have. A family function "*" is defined. When two POINTs are to be multiplied, the first form in the declaration of the family function "*" which can be matched determines the actual procedure to be used. In this case, if at least two out of three of the parameters and the result are of type POLAR then POLAR multiplication is used. (Note that a RECTANGULAR is in the family POLAR but not in POLAR **only.** A RECTANGULAR may be coerced to a POLAR if necessary.) Otherwise RECTANGULAR multiplication is used. Note that the final form can always be matched.

```
family POINT is RECTANGULAR including (POLAR only);
family function "*" (P1, P2: POINT) return POINT is
 match
  when form (POLAR only, POLAR only) return POLAR
   or form (POLAR, POLAR only) return POLAR only
   or form (POLAR only, POLAR) return POLAR only ⇒
   function "*" (P1, P2: POLAR) return POLAR is
    begin . . . end;
   when form (RECTANGULAR, RECTANGULAR) return
                                        RECTANGULAR ⇒
   function "*" (P1, P2: RECTANGULAR) return RECTANGULAR is
    begin . . . end;
 end match;
```

Fig. 6 Example of a family function

Only in rare cases (for example
POLAR_1 := RECTANGULAR_1 * (POLAR_2 * RECTANGULAR_2)
where the intermediate result may be of either type) is qualification required. The interested reader is referred to Ichbiah *et al.* (1979, Section 7-5).

So far we have restricted our attention to very simple examples. Data independence is likely to be most useful in more complicated situations.

Consider a geographical information system where a logical type (family) CURVE is to be used for approximations (to a fixed precision) to curves. There are a number of possible ways to represent curves. A polygonal representation is a possible universal representation. However, significant savings may be realised if special purpose representations are allowed for circles and other special curves. In addition, more complicated representations may be desired to speed processing in certain situations. For example, in some cases the time required to determine whether a point is inside an irregular n-gon can be reduced from $O(n)$ to $O(1)$ by using a suitable representation (Burton, 1977). Raster representations also may be desirable, for example if a curve is to be frequently displayed on a raster device. Compound representations may also be defined. For example, if a circle is to be frequently used in situations where a simple point–radius representation will speed processing and also is to be frequently used in situations where a more complicated representation is required, then a compound representation containing both simple representations may be defined. Other representations may also be justified. An ad hoc solution to data independence in geographical information systems has been proposed (Burton, to appear).

The Ada extension considered earlier can clearly cope with problems such as permitting multiple representations for curves. The proposed language extension will support a flexible and extensible system as well as allow a user to work with logical rather than physical types.

Suppose a new operation is to be added to the system. In the first instance, it is only necessary to support one version of the operation. Coercion is used when parameters are of the wrong type. Similarly, if a new physical type (representation) is to be added to the system then in the first instance it is only necessary to define one or two conversion operations. In general, if a system supports m binary operations and n physical types then only $m + 2(n - 1)$ routines are required (assuming coercion from the parent type to its subtypes is always supported), as opposed to the mn^3 routines which would be required when providing a routine for each possible case.

To ensure system efficiency, instrumentation may be used to detect common cases and additional routines may be added to avoid bottlenecks caused by excessive conversions. These alternative routines will be inserted within the family declaration and automatically utilised (if appropriate) when a package uses the family functions concerned.

5. A view of other language features

The fact that a subtype of a given type may be restricted in use rather than in value means that constants of a type can be thought of as belonging to a constant subtype. For example, with the definition of CONSTANT_PACKAGE in **Fig. 7,** we can declare

```
generic (type T; INITIAL:T)
package CONSTANT_PACKAGE is
 restricted subtype CONSTANT_T is T by private;
 VALUE: CONSTANT_T;
 function "T" (CT: CONSTANT_T) return T;
private
 restricted subtype CONSTANT_T is T by T;
 VALUE: CONSTANT_T:= INITIAL;
 function "T" (CT: CONSTANT_T) return T is
  begin
   return CT;
  end;
end CONSTANT_PACKAGE;
```

Fig. 7 Definition of the generic package CONSTANT_PACKAGE

```
package FIVE is new CONSTANT_PACKAGE (INTEGER,
5);
```

making FIVE.VALUE an INTEGER constant with value 5 for all practical purposes. That is, we can use FIVE.VALUE in any situation where an INTEGER value is required since coercion to INTEGER is allowed. However, there is no possible way to modify the value of FIVE.VALUE.

We do not advocate the use of the generic CONSTANT_PACKAGE but believe that viewing constants in terms of subtypes is more satisfactory than the alternatives, such as the Algol 68 (van Wijngaarden *et al.*, 1969) view of variables as references (pointers) to constants. Constants are simply objects for which the assignment operation is not defined.

If we view a function or an expression as returning an object with a constant subtype then we no longer need special syntactic rules to prohibit functions and expressions from occurring on the left hand side of an assignment statement. Assignment to the result of a function or expression evaluation would be an invalid operation for an object having a constant subtype.

This view can be extended to simplify and clarify other aspects of programming languages. Functions can be viewed as constant arrays. This parallel suggests useful language extensions. An array, being a 'variable function', could be initialised by a function body, and sparse arrays could be supported along the lines advocated by Dahl *et al.* (1972). In this way we could define a sparse array SQUARE_MAYBE, whose domain is all INTEGERs, where SQUARE_MAYBE(I) is I*I unless specifically set to some other value. A possible declaration for SQUARE_MAYBE is given in **Fig. 8.** In Ada

```
SQUARE_MAYBE: array (I: INTEGER) of INTEGER :=
 begin
  return I*I;
 end;
```

Fig. 8 Definition of sparse array SQUARE_MAYBE

it is possible to specify packing for a type. In a similar manner, a sparse specification could be permitted.

User defined selectors may be supported in a language. For example, T.LEFT_MOST_LEAF could be used to select the left most leaf of a tree T, where LEFT_MOST_LEAF is a user defined selector. Selectors are simply functions which are not restricted to return constants. Recall that an array subscript is actually a special type of selector for arrays. We do not particularly advocate the provision of user defined selectors but note that such a facility would fit neatly into the framework we have described.

Relations, in the relational data base sense (Date, 1975), are a useful data type. If relations are to be provided, we can view a relational type as a subtype of a set-of-records type. An array type can be viewed as a subtype of a relational type, and the hierarchical relationship between functions and arrays remains unchanged.

Even if languages are not to be enlarged to include features mentioned in this section, we feel that languages should not hide relationships between constants and variables and parallels between functions and arrays. This basic problem of referring to information in a more uniform way has been considered by others (Geschke and Mitchell, 1975).

6. Conclusion

We have described a general subtype construct for use in high level languages. The construct is useful because error checking is improved by associating arbitrary constraints with subtypes, readability is improved by permitting type definitions to more fully describe types and by permitting relationships between types to be more formally defined, and efficiency is improved by providing a more flexible language which allows logically similar objects to have physically different representations.

It has been shown that with some additional language features a high degree of data independence may be realised in a general purpose high level language.

Finally, we have seen that by thinking in terms of types and subtypes it is possible to clarify and unify the distinctions between constants and variables, functions and arrays, arrays and relations, and so forth.

References

ACM (1979). Preliminary Ada reference manual, *SIGPLAN Notices*, Vol. 14 No. 6, part A.

ANSI (1966). *American standard FORTRAN*, ANSI X3.9, American National Standards Institute, New York.

BURTON, W. (1977). Representation of many-sided polygons and polygonal lines for rapid processing, *Communications of the ACM*, Vol. 20 No. 3, pp. 166–171.

BURTON, W. (1979). Logical and physical data types in geographical information systems. *Geo Processing*,Vol. 1, pp.167-181.

DAHL, O.-J. DIJKSTRA E. W. and HOARE, C. A. R. (1972). *Structured Programming*, pp. 148–155. Academic Press, London.

DATE, C. J. (1975). *An Introduction to Database Systems*, Addison-Wesley, Reading, MA.

GESCHKE, C. M. and MITCHELL, J. G. (1975). On the problem of uniform references to data structures, *IEEE Transactions on Software Engineering*, Vol. 1 No. 2, pp. 207–219.

ICHBIAH, J. D., BARNES, J. G. P., HELIARD, J. C., KRIEG-BRUECKNER, B., ROUBINE, O. and WICHMANN, B. A. (1979). Rationale for the design of the Ada programming language, *SIGPLAN Notices*, Vol. 14 No. 6, part B.

JENSEN, K. and WIRTH, N. (1974). *Pascal: Uses Manual and Report*. Springer, Berlin.

WIJNGAARDEN, A. van, MAILLOUX, B. J., PECK, J. E. L., and KOSTER, C. H. A. (1969). *Report on the Algorithmic Language ALGOL* 68. Mathematisch Centrum, Amsterdam.