Type Extensions

N. WIRTH
Institut für Informatik, ETH, Zurich

Software systems represent a hierarchy of modules. Client modules contain sets of procedures that extend the capabilities of imported modules. This concept of extension is here applied to data types. Extended types are related to their ancestor in terms of a hierarchy. Variables of an extended type are compatible with variables of the ancestor type. This scheme is expressed by three language constructs only: the declaration of extended record types, the type test, and the type guard. The facility of extended types, which closely resembles the class concept, is defined in rigorous and concise terms, and an efficient implementation is presented.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*data types and structures*; *modules, packages*; D.3.4 [**Programming Languages**]: Processors—*code generation*

General Terms: Languages

Additional Key Words and Phrases: Extensible data type, Modula-2

1. INTRODUCTION

Modern software development tools are designed for the construction of extensible systems. Extensibility is the cornerstone for system development, for it allows us to build new systems on the basis of existing ones and to avoid starting each new endeavor from scratch. In fact, programming is extending a given system.

The traditional facility that mirrors this concept is the *module*—also called *package*—which is a collection of data definitions and procedures that can be linked to other modules by an appropriate linker or loader. Modern large systems consist, without exception, of entire hierarchies of such modules. This notion has been adopted successfully by modern programming languages, such as Mesa [4], Modula-2 [8], and Ada [5], with the crucial property that consistency of interfaces be verified upon compilation of the source text instead of by the linking process. An enormous number of calamitous pitfalls, which constituted a genuine impediment to the construction of extensible systems, have thereby been eliminated. The essential ingredient of these systems is that a new module can be developed and changed without the need for recompilation of the modules on whose resources it rests. It was made possible by the textual separation of a module description into a definition and an implementation part, where the former specifies the interface.

Author's address: Institut für Informatik, ETH-Zentrum, CH-8092 Zurich, Switzerland.

Extensibility has, however, been largely confined to procedural aspects. The aspect of applying extensibility to data, and data *types* in particular, has received less attention. The first successful step in this direction appears to have been the concept of the *class* in Simula 67 [1], and in particular the notion of the prefix to a class. It has been adopted in derivatives of Simula, notably Smalltalk [2, 3], Object Pascal [7], and C++ [6].

The basic idea is to provide a facility for defining a new data type that relates to an existing type by some rule of compatibility. It receives genuine importance if the new type can be defined in a module importing the existing type, that is, when the class feature coexists with that of separately compilable modules. Regrettably, the class feature has remained rather vaguely defined and is usually presented in combination with other language features. This factor is witnessed by the undue length and verbosity of language manuals. The concept has therefore remained difficult to understand and apply.

The main purpose of this paper is to base data type extensions on well-understood mathematical concepts and to present a rigorous, complete, and concise definition.

2. EXTENDED DATA TYPES

The notion of extensible data type to be presented here is based on the well-known concept of data type used in most modern programming languages. Here we shall in particular adhere to the syntax of Modula-2. It is based on the record structure, the cornerstone of programmer-defined data types.

Definitions. Given a declaration of the form

$$T' = \text{RECORD } (T) \ldots \text{END}$$

T′ is a (direct) *extension* of T, and T is the (direct) *base type* of T′. We write $T' \rightarrow T$.

Let, for example, a type T and extensions T0 and T1 be defined as

T = RECORD *x*, *y*: INTEGER END
T0 = RECORD (T) *z*: REAL END
T1 = RECORD (T) *b*: BOOLEAN; *s*: CHAR END

The base type is extended by the record fields specified in the declaration of the extension. The components (fields) of an extended record type are therefore those specified in its declaration *in addition* to those of the (direct) base type. In the given example, T0 has fields *x*, *y*, *z*, and T1 has fields *x*, *y*, *b*, and *s*. It is essential that the definition of the extension need not occur in the same module as that of its base type. It can occur in any module importing the base type.

Extended types can be reextended, thereby giving rise to type hierarchies. For example, let T00 be defined as

T00 = RECORD (T0) *w*: LONGREAL END

and hence having fields *x*, *y*, *z*, and *w*.

Definitions. A type T′ is an *extension* of a type T if T′ = T or T′ is a direct extension of an extension of T. T is a *base type* of T′ if T = T′ or T is the direct base type of a base type of T′. We write T′ ⇒ T.

Using this notation, the relationship of the previously declared types is denoted as

$$T00 \rightarrow T0 \rightarrow T,\ T1 \rightarrow T, \qquad T00 \Rightarrow T$$

Intimately coupled with the concept of data type is that of *assignment compatibility.* The conventional rule is that the type of the assigned value x must be the same as that of the variable v to which x is assigned. It is, however, quite appropriate to consider an instance (variable) of an extended type to also be an instance of its base type(s). Accordingly, it is appropriate that a value x be assignable to a variable v if the type of x is an extension of the type of v. Hence we generalize the compatibility rule type(x) = type(v) to type(x) ⇒ type(v).

This generalization, however, requires some closer inspection. A record type represents the Cartesian product of its component types. Each record value is mirrored by a point in the Cartesian space in which each dimension corresponds to a record component. A base type is therefore represented by a space of lower dimensionality than those of its extensions, and hence an assignment to a variable of a base type must be regarded as a *projection* onto the Cartesian space spanned by the base type.

Definition. An assignment $v := x$ is possible if the type of x is an extension of the type of v, that is, type(x) ⇒ type(v). It constitutes a projection of x onto the type of v.

Let us, for example, assume the following declarations of variables:

v: T
$v0$: T0
$v1$: T1

Then the following assignments are possible:

$v := v0$
$v := v1$
$v := v00$
$v0 := v00$

The assignment $v := v0$, for example, consists of the components $v.x := v0.x$ and $v.y := v0.y$. The field $v0.z$ remains uninvolved. Hence all values with identical components x and y result in the same value v. $v0$ is projected onto the x–y space. In contrast, the assignments

$v0 := v$
$v1 := v$
$v0 := v1$

must be rejected because the assigned values do not completely specify a value of the respective assignee. It is tempting to consider the instances of a type as a set characterized by the type. Then the instances of an extended type appear as a subset. However, the "merging" of different elements of the subset into the

same element of the superset, as may happen in the case of projections, is inconsistent with the set notion in which distinct elements retain their identity when being removed from the subset.

3. DATA STRUCTURES WITH EXTENDED TYPES

The concept of related types becomes genuinely useful in connection with data *structures,* and in particular with dynamically generated structures, because it permits the construction of heterogeneous structures. They typically represent the relationship among their elements by pointers. In typed languages, pointers are always bound to a particular type called the *pointer base type.* It is therefore necessary and logical to extend the concept of type extension to pointer types.

Definition. A pointer type P0 is an extension of a pointer type P if its pointer base type is an extension of the pointer base type of P.

Examples are

```
P   = POINTER TO T
P0  = POINTER TO T0
P1  = POINTER TO T1
P00 = POINTER TO T00
```

$$P00 \rightarrow P0 \rightarrow P,\ P1 \rightarrow P, \qquad P00 \Rightarrow P$$

The stated rule of assignment then covers the case of pointers too. In analogy to the record assignments shown above the pointer assignments

```
p := p0
p := p1
p := p00
```

are possible, whereas

```
p0 := p
p1 := p
p0 := p1
```

are not. The examples demonstrate that the assignment rule guarantees that a pointer variable bound to a base type T may possibly refer to a variable whose actual type is a (genuine) extension of T but never to a variable whose type is not an (extension of) T.

Because of the relaxation of the assignment rule, it is possible to construct data structures that are heterogeneous, that is, structures whose elements are of different types. However, they are all related by having a common base type. Assume, for instance, a tree structure based on the following declarations where T denotes the node type

```
P = POINTER TO T
T = RECORD key: INTEGER;
      left, right: P
    END
```

The elements may be of any extension type of T because the pointers referring to them are extensions of P and can therefore be assigned to the linking fields *left* and *right.*

Procedures for manipulating the *structure*, such as insertion, deletion, and search, are defined in terms of the (base) type. Procedures manipulating *individual nodes* will be specific for each extended type. This facility is once again particularly attractive if the base type together with the structure-manipulating procedures is defined in a (base) module, whereas its client modules individually define extensions of the base type together with procedures operating on respective nodes. New client modules with new extensions may be added to a system at any time without affecting the base module or other client modules or even requiring their recompilation. This clearly contrasts with the notion of the abstract data type in which all applicable operators are supposed to be defined *together* with the data type.

An example of a dynamic, heterogeneous data structure is taken from system software: Let the base type be *Window*, with extensions *TextWindow*, *GraphWindow*, and *PictureWindow*. Another example stems from a compiler whose "symbol table" is a dynamic data structure with elements of the base type *Object*. Appropriate extensions are *ConstantObject*, *VariableObject*, *TypeObject*, and *ProcedureObject*. Extensions of *ConstantObject* are, for instance, *IntegerObject*, *RealObject*, and so forth. Again, we emphasize the fact that the extensions can be declared in different modules, which may be added to a system without invalidating the existing modules.

4. TYPE DISCRIMINATION

If, for example, after a search through a tree, an operation is to be applied to a referenced element, it may well be necessary to determine the actual type of that element since the operation may depend on the particular extension type. However, the element is referenced by a pointer obtained from the structure (fields *left* and *right*). The pointer is therefore of the (common) base type and does not yield access to extension specific information. In order to access fields of extensions it must be established that the element is actually of the anticipated type. A facility for type discrimination is therefore required. We emphasize that this is in contrast to the case of statically defined variables since their (actual) type is always the one explicitly declared in the program text.

Definition. A *type test* is a Boolean expression of the form v IS T, where v is a designator and T a type. The symbol IS is classified as a relational operator and is pronounced as "is (of a type which is) an extension of."

Because types must be identifiable within variable designators, and since designators do not include expressions, a second form of discriminator is suggested. Its sole purpose is to guarantee that a designator is of the indicated type.

Definition. A *type guard* applied to the designator v has the form $v(T)$. It asserts that v is of type T. If this is not so, the guard aborts execution of the program.

Referring to the preceding examples, we recall that the assignment $v0 := v$ is not admissible. It is, however, if an appropriate guard is applied to v, namely $v0 := v(T0)$. By the same token, access to the field z via v, that is, the designator

$v.z$, is not admissible. However, the designator $v(T0).z$ is correct since the guard asserts that v is an object of type T0, which has a field z.

5. EXTENDED TYPES VERSUS VARIANT RECORDS

Inhomogeneous data structures are quite common. Most languages offer some facility to express these structures in tricky and, from a mathematical point of view, unsatisfactory ways. In Pascal and Modula-2, they are expressed as variant records. Specific variants are designated by values of a (syntactically) distinguished field called the *tag field.* The variant record facility bears several drawbacks. Of particular relevance is the danger of misusing a field declared as belonging to one category, whereas actually the tag value specifies the individual record to be of another category. This can have catastrophic consequences in the case of assignment to the field. In principle, every field access could be preceded by an (implicit) check of the tag value validating the field in question. Most implementations forego this "overhead." An even worse misuse of the facility is the assignment of a new value to the tag field istelf: As a side effect, it invalidates all previous assignments to the fields of the specified category.

A severe handicap is the fact that further cases (variants) cannot be added unless the record declaration is altered. Therefore, the module containing it must be recompiled, and with it all the client modules referring to it. The variant record facility clearly defies the notion of extensible system design.

In systems relying on automatic storage retrieval (garbage collection), the variant record is a severe stumbling block. Its handling is complex and cumbersome to say the least; we note in passing that a declaration may contain several discriminated cases at the same level, and also that they may be nested. It is practically impossible to use a garbage collector in conjunction with variant records without restricting the feature in some way.

In contrast, the presented facility for type extensions allows for a safe and efficient implementation, in particular in conjunction with automatic storage retrieval.

6. POLYMORPHIC PROCEDURES

The adjective polymorphic is used for procedures that feature parameters for which the strict type consistency rule (equality of formal and actual parameter types) is relaxed. We note that the presented concept of extended types includes precisely such a relaxation. If a formal value parameter is of base type T, then the corresponding actual parameter can be of any extension of T. This follows from the fact that value parameters are considered as local variables to which the actual parameter's value is assigned initially.

Reference parameters (in Pascal and Modula called VAR-parameters) are considered as local pointers to which a reference to the actual parameter is assigned initially. It follows that the same relaxation holds for both value and reference parameters.

7. TYPE EXTENSION AND INFORMATION HIDING

The most important aspect of the module concept from the point of view of system design is the decoupling of individual modules. By this we refer to the

possibility of developing modules separately, once their interfaces are defined. More specifically, it is essential that changes in subordinate modules be accomplished without the need for changing their client modules. This includes recompilation of subordinate modules without the need for recompilation of the clients.

This decoupling is achieved by separating a module description into a *definition part* and an *implementation part.* Changes to a module remain hidden, as long as the definition part remains unchanged. The definition part constitutes the interface in which the name only of a type is declared, whereas its properties and the applicable operations are specified in the implementation part. A type defined in this manner is called an abstract data type. In Modula-2 the type is called *opaque*, and it effectively constitutes a pointer type whose binding is specified in the implementation part.

Sometimes the total invisibility to clients appears as a harsh measure, and it is circumvented by the introduction of procedures merely inspecting a single component of the hidden data structure. A more general solution consists of the declaration of the visible part of a type in the module's definition part and of its extension in the implementation part. Let, for example, a type T be declared in the definition part of a module M:

```
T = RECORD
      x, y: INTEGER
    END
```

and then let T be extended in M's implementation part by the private fields a and b (for convenience, the declaration of x and y is repeated):

```
T = RECORD
      x, y: INTEGER;  (*externally visible fields*)
      a, b: INTEGER   (*private fields*)
    END
```

An explicit mention of the base type is unnecessary because the (re)use of the name T declared in the definition part identifies this (re)declaration as an extension.

We draw attention to the fact that extension of a type specified in a module's definition part is of a conceptual nature only. Referring to the example, every variable of type T consists of the four components x, y, a, and b; nonextended versions do not exist. It is therefore appropriate to regard the declaration given in the definition parts as a *public projection* of the complete declaration given in the implementation part.

8. IMPLEMENTATION

The simplicity of the presented type extension concept raises the hope that its implementation is simple too and therefore also efficient. This is indeed so; in particular, the addition of type extensions does not impinge on any of the basic facilities already present in languages such as access to record fields. The only new constructs requiring additional instructions are the type test and the type guard. Since they do not occur with significant frequency, at least in soundly designed programs, they contribute little to execution time. It is noteworthy that the concept can be implemented entirely free of any run-time support routines. Type test and type guard result in a small number of in-line instructions.

When considering a way to implement the type test, we notice that the type of a variable can be determined by a mere textual scan, except when the variable is accessed indirectly, that is, in the cases of pointers or VAR-parameters. Only in these two cases need a compiler generate additional instructions and need the variables be supplemented by additional type information.

Variables accessed via pointers are (usually) allocated dynamically, for instance, by an explicit executable statement NEW(p) that sets aside the storage and assigns its address to the pointer variable p. It is at this point that the variable is supplemented with an identification of its type. In passing we note that systems relying on automatic storage retrieval (garbage collection) require a type description anyway. In practice this identification consists of a (hidden) pointer to a type descriptor. The pointer is called a *type tag*.

In the case of (record typed) VAR-parameters, the type tag of the referenced variable is supplied together with its address. This is necessary because the actual record may be statically allocated, and therefore, does not carry a tag.

For our purpose, type descriptors are extended by an additional field representing a pointer to the descriptor of the type's base type (the *base tag*). This information is used to evaluate type tests and to execute type guards. In order to explain the required operations, we postulate as an example the following constellation of types:

$$\text{T000} \rightarrow \text{T00} \rightarrow \text{T0} \rightarrow \text{T}, \quad \text{T001} \rightarrow \text{T00}, \qquad \text{T01} \rightarrow \text{T0}, \quad \text{T1} \rightarrow \text{T}$$

Consider now the type test v IS T0, where v is a VAR-parameter of type T. (An analogous case is p IS P, where P is a pointer type bound to T). Evidently, a static (compile-time) test can verify that $T0$ is indeed an extension of T. If this were not the case, a programming error would be likely. It is, for instance, obvious from the program text that $v0$ IS $T1$ is false, and hence this situation must be diagnosed as a type error.

The test at execution time must determine whether or not v is an extension of T0, that is, a T0, T00, T01, T000, or T001. This is accomplished by successively comparing the tag T0 with the types (tags) along the linked list starting with the element designated by v ($p\uparrow$). If, in this example, v ($p\uparrow$) is an element of type T0, a match is found by the first comparison, if it is an element of type T00 or T01, it is found by the second comparison, and if it is a T000 or a T001, it is found by the third comparison. If v is of type T1, no match is found since T1 does not lie on the traversed branch of the tag tree.

The type test is expressed by the following piece of program. t is a local variable (register):

```
t := v.tag;
LOOP
  IF t = T0 THEN EXIT (TRUE) END;
  t := t↑.basetag;
  IF t = NIL THEN EXIT (FALSE) END
END
```

The operation for the type guard is virtually identical, except that no Boolean value need be generated, and program execution may be aborted instead.

Since both type test and type guard involve sufficiently few instructions, they can well be represented by in-line code, enhancing the resulting efficiency.

Unfortunately, the provision of a loop construct for descending the linked list of tags is unavoidable since in the context of the test no information is available about the number of extensions of the tested type that may have been defined in other modules. (Note that T00, T01, T000, and T001 may have been defined in modules importing T0). However, in practice, the length of the list of tags will be very small, ensuring that the test will never involve the execution of a large number of instructions.

The implementation of the extended declaration in implementation parts (public projection in the definition part) poses the following problem: When compiling a client module importing the record type, the compiler must be provided with information about the type, in particular its size. This information must therefore be included in the symbol file generated when the definition part is compiled. At this time, however, the size is unknown, if the type is to be extended in the corresponding implementation part. In fact, it must be unknown, because by definition clients must not depend on the imported modules' implementations. We suggest circumventing this intrinsic dilemma by providing a *compiler hint* in the definition part, which indicates a maximum size that the record declaration in the implementation part may require. This hinted size is then taken as the actual size, and respecifications of the type in the implementation part will not affect clients, as long as the required size does not exceed the hinted size.

9. ADDITIONAL PROGRAMMING CONSTRUCTS

The facilities presented here, namely the type extension declaration, the type test, and the type guard are logically sufficient to construct and use inhomogeneous data structures with the guarantee of full type integrity. Nevertheless, one is tempted to suggest additional language constructs with the goal of improving efficiency. An obvious inefficiency of programs restricted to the basic facilities lies in the repeated (and therefore unnecessary) execution of the same checks.

A sequence of references to fields of the same (extended) record require repeated checks. They can be avoided by (textually) binding the record variable to the extended type. For example, given the parameter v of type T, the three statements

$$q := v(\mathrm{T1}).b; \quad ch := v(\mathrm{T1}).s; \quad r := v(\mathrm{T1}).u$$

could be condensed into a construct of the form

```
WITH v: T1 DO
  q := v.b; ch := v.s; r := v.u
END
```

Here the WITH clause merely states that v is to be considered as of type T1 within the contained statement sequence, and that this fact is asserted before execution of the statements. (Note that this kind of WITH statement differs from that of Modula-2: Neither is an anonymous variable involved, nor is a new scope containing the field identifiers opened.) If type guards as previously introduced are considered as punctual guards, this WITH construct appears as a *regional guard.*

Obviously, a conscientious programmer will ensure that a guard will never lead to program abortion. Hence, constructs of the form

IF v IS $T0$ THEN ... $v(T0)$ ... END

will be frequent. This situation is analogous to

IF $p \neq$ NIL THEN ... $p\uparrow$ ... END

with every pointer dereferencing operation implying a guard against the value NIL in spite of the fact that a nonnil value had been asserted by the preceding test. The idea of combining test and guard(s) therefore suggests itself. It might be expressed by a construct similar to the WITH statement

```
WHEN v IS T0 THEN
  q := v.b; ch := v.s; r := v.u
END
```

implying that the statements are executed if the test is affirmative. However, ample experience has taught that the combination of different objectives in a single construct is often of dubious value in practice, causes considerable (and sometimes unforeseen) complications of implementation, and quickly calls for a (nonterminating) sequence of extension proposals. In this case, an ELSIF option, as well as the admission of unrestricted Boolean expressions in the WHEN clause, immediately appear as desirable.

If the temptation to introduce such combinational constructs is resisted, it might be argued that the burden of avoiding double checks could be placed onto the compiler. An "intelligent" compiler might accumulate sufficient contextual information to recognize the superfluity of type guards when they are implied in an IF clause. In these cases, the compiler would suppress the emission of the unnecessary guard instructions. The required compilation scheme would in fact be similar to techniques employed for other code optimizations (i.e., improvements).

It remains an open question, however, whether the resulting gain in execution speed would justify either the introduction of combinational language constructs or a complex additional optimization machinery in the compiler.

ACKNOWLEDGMENTS

The presented concept has emerged from many fruitful discussions with J. Gutknecht. I gratefully acknowledge his insights and suggestions, as well as his acrimonious criticism, if an earlier proposal did not fully satisfy conceptual consistency *and* practical needs.

REFERENCES

1. BIRTWISTLE, G. et al. *Simula Begin.* Auerbach, Pennsauken, N.J., 1973.
2. GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.
3. KAEHLER, T., AND PATTERSON, D. A small taste of Smalltalk. *BYTE*(Aug. 1986), 145–160.
4. MITCHELL, J. G., MAYBURY, W., AND SWEET, R. Mesa language manual. Xerox PARC, CSL-79-3, Xerox, Inc., Palo Alto, Calif., 1979.

5. REFERENCE MANUAL FOR THE ADA PROGRAMMING LANGUAGE. US Dept. of Defense, Washington, D.C., July 1980.
6. STROUSTRUP, B. *The Programming Language C++*. Addison-Wesley, Reading, Mass., 1986.
7. TESLER, L. Object Pascal report. *Structured Language World 9*, 3 (1986).
8. WIRTH, N. *Programming in Modula-2*. Springer-Verlag, New York, 1982.

Received February 1987; revised September 1987; accepted January 1988