DATA TYPES AND EXTENSIBLE LANGUAGES

Philippe JORRAND
IBM France Scientific Center. Grenoble

Definition mechanisms for extensible programming languages, in principle, allow variation of the language definition in various directions: data types, operations, control structure and syntax. We will focus our attention here on the definitional capabilities that one might expect from a data type extension mechanism. First, we will define informally what the nature of a data type is, what kind of information it contains. Then, we will discuss the most classical approach for data type definition in extensible languages. Finally, we will sketch a model for a mechanism which would permit to introduce really new data types in a language.

1. WHAT IS A DATA TYPE ?

Classically, in most programming languages, if not in all of them, the role played by data types is two-fold:

A- They serve to specify the external and internal representations of objects.

B- They serve to define the rules in handling those objects.

An example will help to clarify those two points: let us take the type integer in Algol 60.

A- If one writes the number "100", those characters are recognized as denoting an object which is an integer value, the type information being implicitly associated with the form of that external representation of the object. Then, an internal representation will be built for that object: for example, a fixed point binary representation occupying one word in memory. Thus, one might consider that a "representation function" is associated with the type integer: that function recognizes external representations of integer values and builds corresponding internal representations.

This can be generalized and stated in a slightly different way: given a type t, it serves to specify an external representation for a set of objects and it establishes a correspondence between that external representation and an internal representation of the same objects. However, one may remark that the existence of an external representation is not mandatory. When it does not exist, the type t specifies only the internal representation. For example, this is the case for names (objects of type ref) in Algol 68. (I am not sure whether the converse is meaningful, except, may be, for comments in a program...)

B- An object of type integer, to keep the same example in Algol 60, can be handled in a number of well defined ways: it can be added to another object of the same type, it can be used as a subscript, it can be converted into an object of type real, etc. But it may not be "anded" with an object of type boolean, it may not be subscripted, etc. Thus, there are rules which must be obeyed when using such an object. Those rules are dictated by the fact that the object has the

type integer. Moreover, the user may define other, more elaborate rules for using integer values, by declaring procedures taking parameters of that type. In fact, all possible uses of integers can be considered as defined by functions: a function for integer addition, a function for array subscripting, a function for integer conversion into real, etc. But no function for "anding" an integer with a boolean.

Thus, in general, a data type t serves to define the properties, or the rules in handling all objects having that type t. Those properties can be viewed as specified by the set of functions to which objects of type t can be passed as arguments.

2. CLASSICAL DATA TYPE EXTENSION

In the majority of currently proposed extensible languages, the data type definition mechanisms correspond very closely to a scheme originally described by Standish [2]. That scheme has even been used in other languages which were not explicitly presented as being extensible, for example in Algol 68 [3]. Roughly, this classical way of extending data types works as follows:

A- A number of data types are provided as primitives.

B- The mechanism itself is formulated as a set of constructors, which can be regarded as specialized functions taking a number of existing -i.e. primitive or already defined- data types as arguments and returning a new one as result.

Usually, a declarative statement is also part of that mechanism, thus permitting to identify a newly built data type.

However, that approach for data type definition presents a number of major drawbacks when it is considered in the general framework of an extensible language.

A- With the classical data type extension scheme, a representation function is associated -implicitly- with each primitive data type, exactly in the same way as for data types in non extensible languages. The essential implication is that a user may not choose the representations he wants for objects of primitive data types.

B- The elementary properties of objects of primitive data types are predefined. For example, the conversion scheme between primitive data types is not defineable from within the language.

Thus, as a consequence of points A and B, the number of primitive data types is fixed and predefined. So is their nature. There are no means of putting together a representation function and a number of properties, and of saying that they constitute a data type: one cannot really "create" a new data type which would be at the same level as the primitive ones.

C- The nature and the number of constructors are also fixed and predefined. They imply an internal representation which cannot be chosen by the user. They imply a number of properties for objects having a constructed data type, and one is not free to attach additionnal properties of the same kind.

D- All data types are necessarily related to each other in a strict hierarchical fashion, as specified by the constructors: given a set of data type definitions, all relations between data types are "vertical" and can indeed be represented by a tree-like structure. There is no way of specifying any "horizontal" relation between data types: such relations would include essentially conversions of any kind and transformations between data types, thus generalizing the simple mappings obtained through the hierarchical scheme.

Let us take a very simple example in Algol 68 to illustrate those remarks. The mode -i.e. the type, in Algol 68 terminology- complex is declared by:

```
mode complex = struct( real rp, real ip )
```

If one declares:

```
complex z
```

he may write:

```
z := 3.14
```

That statement assigns 3.14 to rp of z and 0.0 to ip of z. But, if another mode is declared, to represent the mode of imaginary numbers:

```
mode imag = struct( real ip )
```

and if one writes:

```
imag i
```

the assignment statement:

```
z := i
```

is not allowed, and there is no way of making it legal, since it is not possible to define a new conversion transforming imag values into complex values, whereas a conversion was predefined for real into complex.

Clearly, that classical data type definition mechanism is not flexible enough for extensible languages. The reason seems to be that all data types are defined in terms of a number of other simpler preexisting data types. More flexibility and more generality would be provided if the points of departure were more elementary, thus permitting to build really new data types, for which the building blocks would represent the basic information that data types actually contain: the representation functions and the various properties of objects.

3. DATA TYPES: ANOTHER APPROACH

The approach which is very roughly sketched here constitutes a model for a data type definition facility which attempts to provide the sufficient flexibility of definition required in the framework of an extensible language.A more detailed description of that mechanism is presented in [1].

The fundamental assumption upon which that method is built can be stated as follows: a data type is a class of data items having a number of properties in common. In fact, the word "class" will be used instead of "data type".

In that scheme, no class is present at the outset: thus, the base language is a "type-less" language, which acts as an interface between languages where classes have been introduced via extensions, and the machine. The base language provides essentially functions for the representation of data items, whether they belong to a particular class or not: storage access, address calculation, allocation functions, transformation of external representations of values (i.e. constant denotations) into internal representations, operations on internal representations, etc. The base language provides also an elementary function definition an application capability which is close to lambda calculus.

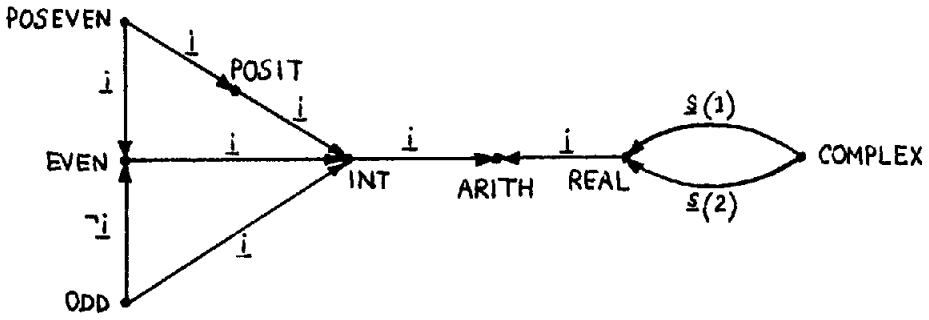
Then, the class definition mechanism is capable of specifying the following points:

A- Definition of a new class, considered as a set, and, eventually, definition of its relations with other classes, using set operations: inclusion, union, intersection, complement and cartesian product. When a new class has been introduced it has a purely formal existence, since, in general, no element (data item) belongs to it nor any functional property has been defined for its potential elements.

In fact, defining a class can be viewed as introducing a node in a graph, and set operations specify various kinds of arcs between the nodes. For example:

```
def( INT, class )
def( EVEN, in INT )
def( REAL, class )
def( ARITH, union( INT, REAL ) )
def( POSIT, in INT )
def( POSEVEN, inter( EVEN, POSIT ) )
def( ODD, compl( EVEN, INT ) )
def( COMPLEX, cart( REAL, REAL ) )
```

Those definitions are pictured by the following graph, where an i-arc means inclusion, a ¬i-arc means non inclusion and an s(i)-arc means selection of the i-th component of a cartesian product:



Inclusion of a class C1 into a class C2 means that the elements of C1 have all the properties of the elements of C2, and that they may have other specific properties. Non inclusion means that the elements of C1 have none of the specific properties of the elements of C2.

B- Properties of all data items belonging to a particular class. These properties may be considered as "rules of utilization" of the data items: they are defined as functions to which the elements of the considered class can be passed as arguments. In fact, the relations between classes which are deduced from set operations are of a very primitive nature. More elaborate relations can be defined between two classes C1 and C2 when it is possible to define functions taking as argument an element of C1 and producing as result an element of C2: such functions define the properties of the elements of C1.

Two kinds of functional relations can be defined between classes: conversions and procedures. The essential difference between a procedure and a conversion is that requests to evaluate a procedure are always explicitly formulated in the program whereas, in general, they are done implicitly for conversions: with a given data type scheme in an extended language, it is the job of the compiler to choose conversions.

Conversions and their classification

A conversion is defined by:

def(N, conv(l, C1, C2, λ-exp))

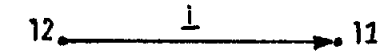
where N is the -optional- name of that conversion, C1 and C2 specify that it is a conversion from class C1 to class C2 and the λ-exp describes the process of conversion itself. The argument l introduces the considered conversion into a set of conversions called a level: by their definitions, conversions are classified into various levels. Levels themselves are defined by:

def(l, level)

and, since a level is a set, set operations like in, union, inter and compl may be used in the definition of levels. For example:

def(l1, level)
def(l2, in l1)

Thus, levels may also be represented in a graph:

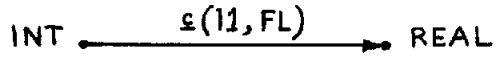


Then, the level 11 being defined, it possible to introduce into that level a conversion from INT's into REAL's:

def(, conv(11, INT, REAL, λI.float(I)))

where float is one of the functions of the base language: it transforms a fixed point internal representation into a floating point internal representation.

Such a definition adds a new arc to the graph of classes:



where FL represents the λ-exp.

Procedures

A procedure is defined by:

def(P, proc(k, C1, C2, λ-exp))

where P is the name of the procedure, C1 and C2 specify that P takes its argument in C1 and returns a result in C2, the λ-exp describes the actions of the procedure, and k is a level of conversions. The evaluation of the argument of P may require a number of conversions in order to produce a value of class C1, and those conversions would be applied to the value which has been originally passed as argument. The level k specified in the definition of P sets a maximum level for the conversions which may be applied during the evaluation of its argument: any conversion belonging to k or to any level k', such that i*(k',k) is true, is allowed. (i* is the transitive closure of the relation i(l',l) which is true when an i-arc goes from l' to l in the graph of levels.)

Thus, the classification of conversions into levels constitutes a formalization and a generalization of the essential concept involved in the notion of "syntactic position" in Algol 68, where conversions -called "coercions" in [3]- obey to a linear hierarchy.

Examples of procedure definitions:

```
def( SQRT, proc( 11, REAL, REAL, some λ-exp S ) )

def( ADD, proc( 12, INT2, INT,
                        λI.fixadd( sel(1,I), sel(2,I) ) ) )

def( ADD, proc( 12, REAL2, REAL,
                        λX.floadd( sel(1,X), sel(2,X) ) ) )
```

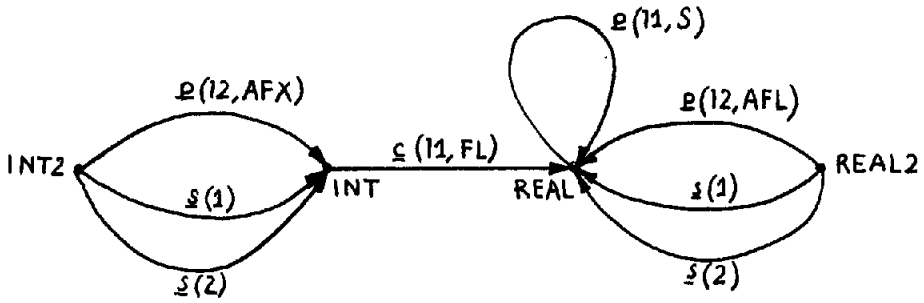
where 11 and 12 are the levels defined earlier and where INT2 and REAL2 are defined by:

```
def( INT2, cart( INT, INT ) )
def( REAL2, cart( REAL, REAL ) )
```

The operator sel performs selection on cartesian products. The functions fixadd and floadd are functions of the base language performing addition on fixed point and floating point binary internal representations respectively.

It must be noted that the procedure ADD is generic, in the sense that two different meanings have been attached to it.

Those definitions add a number of arcs to the graph of classes:



where AFX and AFL represent the λ-expressions respectively specified in the two definitions of ADD.

C- Membership of a particular data item in an existing class. A primitive data object built by an expression of the base language does not belong to any class. But it is possible to define classified objects: their actual meaning still is some internal object built by the base language, but, when they are used, they must obey to the rules attached to the class to which they belong.

A classified object is defined by:

```
def( N, as( C, E ) )
```

where N is its name, C is a class and E is some base language expression. The actual meaning of N, at the base language level, is the result of E, but it will be used as an object of class C.

It must be noted that classified objects are also built by the evaluation of conversions and procedures.

D- Utilization of the graph. All the strategy for data type control and conversion is defined by the graph of classes and the graph of levels. Given a procedure defined by:

def(P, proc(k, C1, C2, some λ-exp L))

and given a classified object A belonging to C1, a call of P with the argument A is represented in the base language by:

eval(bind(L, A'))

where A' is the internal object represented by A, and where bind and eval are two functions of the base language, for the binding of parameters and the evaluation of expressions respectively.

Thus, all class information has disappeared in that base language statement. In fact, that statement is automatically generated after the class checking has been performed: does A belong to the class C1 required by P ?

Class checking and production of the base language statement are done by a special operator, called apply, which constitutes the essential interface between the class mechanism and the base language:

apply(P, A)

If the classes match, the base language statement is produced. Otherwise, the application fails.

But the operator apply does not permit intervening conversions in the evaluation of the argument. Another operator, call, will take advantage of the possible existence of conversions in the graph of classes:

call(P, X)

If X belongs to C1, the effect is the same as apply(P,X). But, if this is not the case, a conversion path must be found in the graph. That path, which goes from the class C of X to C1, is built only with conversion and inclusion arcs (c(l,L)-arcs and i-arcs). But it is possible that more than one path satisfy these conditions. If they differ only by i-arcs, they are considered as a single path, since no action is involved by such arcs: only conversion arcs are of interest. But if they differ by one or more c(l,L)-arc, a choice must be made: this is the main reason why more selectivity is introduced by the levels of conversions. If the definition of P has set a maximum level k for conversions applied to its argument, all c(l,L)-arcs of the path must be such that l=k or i*(l,k) is true. Then, if more than one path remain, the data type scheme which has been defined is ambiguous.

When a single path has been found, a functional composition of all the λ-expressions specified in the implied conversions is applied to the object X' represented by X, and the result of those conversions is passed as argument to the λ-expression specified in the definition of P.

For example, if J is defined by:

def(J, as(POSEVEN, some expression yielding J'))

the base language statement produced by:

call(SQRT, J)

is:

eval(bind(S, eval(bind(FL, J'))))

In the case of:

call(ADD, (I,J))

where (I,J) builds an object of class INT2, there will be no temptation to use the definition of ADD for REAL addition, since ADD has set a maximum level 12 for conversions applied to its argument. Thus, the choice of the correct meaning of ADD is quite simple, and no ambiguity is due to the fact that ADD is generic.

REFERENCES

[1] JORRAND, P. and BERT, D., On some Basic Concepts for Extensible Programming Languages, Proceedings of the International Computing Symposium, Venice, (April,1972)

[2] STANDISH, T. A., A Data Definition Facility for Programming Languages, Ph.D. Thesis, Carnegie Institute of Technology, (May,1967)

[3] van WIJNGAARDEN, A. (Ed.), et al., Report on the Algorithmic Language Algol 68, MR.101, Mathematisch Centrum, Amsterdam, (October,1969)