X3T2/91-072

Objects and Pointers

by Ed Greengrass

This paper can be viewed as a follow-on (and correction) to an observation I made in WG11/N228. It also takes up some related points raised by Yellin's interesting paper, WG11/N221. In both cases, my concern is the content of the Common Language-Independent Data Types (CLIDT) Standard, Working Draft 4 (WD 4), WG11/N190.

My principal conclusions are:

(1) that Object of type T is a legitimate and valuable Generator in the CLIDT sense of the term,

(2) that the characterizing operations of Object include Create, Delete and Assign value. These characterizing operations apply to Object of type T, not to T itself (and not to Pointer to type T either),

(3) that objects, in contrast to values, are uniquely identified by system-assigned Unique Identifiers (UID's). Hence, two objects of type T can have the same value of type T and yet be distinct values of Object of type T,

(4) that mechanisms for assignment of UID's within a given system are implementation-specific and hence outside the realm of CLIDT but that exchange of objects (and of pointers to objects) in conformance to CLIDT requires that UID spaces be outwardly and inwardly mapped one-one from system to system,

(5) that another legitimate Generator is Named Object of type T which generates a value space of objects identified by name rather than UID such that the names are user-assigned (typically by a programmer) rather than system-assigned as UID's are, and finally,

(6) that several other kinds of object identification are already covered by existing CLIDT generators: a Table is a collection of objects such that each is identified by a part of its value called its key, a Set is a collection of objects such that each is identified by its entire value, an Array is a collection of objects such that each is identified by its ordinal position, called its index value.

In the discussion of Pointer subtypes in N228, I stressed that a Pointer, by its very (conceptual) nature, points to an object containing a value (an instance in the usage of CLIDT WD 4) rather than to the value itself. I then raised the question of whether these objects themselves formed the values space of a type and if so what that type was. The question was legitimate but the answer I gave to it in N228 was inadequate.

1. Objects and UID's

What is an "object" of type T? It is a container or slot into which one can legally stuff values of type T. Moreover, it can be uniquely identified by what the object-oriented technology people call (not surprisingly) a Unique ID (UID). The UID is "system-generated", meaning that its value has no intrinsic, application specific, meaning; it is just the value that one can use to access the object it identifies. In essence (as I pointed out in N228), UID's are the value space of type Pointer to Object of type T.

The mechanism by which UID's are generated or used to access objects is an internal implementation detail, outside the realm of CLIDT. What IS necessary is that it be possible to exchange

UID's via CLIDT. It is also desirable that one be able to exchange objects themselves. Note that exchanging an object of type T is NOT the same thing as exchanging a value of type T, a point I discuss below.

2. Exchange of Pointers

What does exchange of Pointers imply? Consider a graphical structure G of objects linked by pointers. Let it be sent from system S1 to system S2. Then, within S1 the pointer values are generated by an S1 mechanism from a UID space defined by S1. When G is received and stored within S2, the pointer values are generated by an S2 mechanism from a UID space defined by S2. During the exchange process, the pointer values may be generated by an exchange service mechanism from a UID space defined by the exchange service. The essential requirement is that the graphical structure be logically the same in S1 and S2 (and hence that the outward and inward UID mappings should be one-one). Hence, one should be able to traverse G in the same way in S2 as one could in S1, using equivalent Dereference operations.

What if G is not an elaborate graphical structure but a simple Pointer to object of type T, such as might be passed as an operand of a procedure call? The same principle applies: One should be able to obtain the current value of the object from the pointer value, in S2 just as easily as in S1. Note that this says nothing about WHEN the object itself is physically exchanged, e.g., it says nothing about whether the object is passed along with the pointer, exchanged separately when S2 dereferences it, exchanged in pieces as needed, etc. These are questions for CLIPCM, RPC, and the underlying implementation mechanisms. if S1 modifies the value of the object between exchange of the pointer and dereference by S2, then the object value S2 receives may be different from the value that the object had when S1 sent it. Such considerations are wholly outside of CLIDT. CLIDT only insists that S2 be able to dereference the pointer in the same way as S1, obtaining the current value of the SAME object. That is what it means to say that Dereference is a characterizing operation of Pointer.

3. Objects vs. Values

Now, what about the object of type T itself? Is it a value of a type from the CLIDT perspective? What is this type? What are its characterizing operations? How does exchanging a value of type T differ from exchanging an object of type T?

3.1 Exchange of Objects

One needs to exchange objects of type Record in order to exchange values of graphic structures such as G, discussed above. Each node of G (except perhaps terminal nodes) would have to be a Record in CLIDT terms because it would have to contain at least two fields: one field for the value of the node and one or more fields for pointers to other nodes. These node records must be objects of type Generated Record rather than merely values of the type because a node N1 might point to two distinct nodes N2 and N3 having the same value, etc. (Note, by the way, that the UID would not be a field of the generated record, i.e., it is not part of the value of the record; UID's are ASSOCIATED with objects, enabling objects to be used and exchanged. The physical mechanism of the association is an implementation detail which may vary from system to system.)

But even in a simple case such as a procedure operand of type Pointer to Integer, the integer must, in general be exchanged as an object. That is because it MIGHT be exchanged separately, e.g., at a later time than the pointer. Hence, there must be a way of associating the pointer value (the UID) with the integer object to which it points.

3.2 Characterizing Operations of Object

What are the characterizing operations of Object of type T, as distinct from the operations on type T itself? In other words, what "added value" does associating UID's with values add to the value space? There are three things you can do with an object (thanks to its UID) that you can't do with a value: create it

(i.e., create an instance of some value V), delete it, and assign a value to it (i.e., change its value).

3.3 Characterizing Operations of Pointer

The characterizing operations of an object should not be confused with the characterizing operations of a reference to an object, i.e., the characterizing operations of Pointer to type T. The two things one can do with a reference to an object are (1) obtain the value of the referenced object (operation Dereference) and (2) compare a reference to object O1 with a reference to object O2 (operation Equal) to determine if they reference the identical object (same UID) or different objects (different UID's).

The distinction between Object of type T and Pointer to object of type T is emphasized by Resolved Issue 14 of CLIDT WD 4 which says that operations like Allocate (which creates both a pointer and an object for that pointer to point to, and Associate which presumably associates an existing pointer with an object, are NOT characterizing operations of Pointer. In terms of the above discussion, Associate would be an assignment operation on an object of type Pointer to T (perhaps preceded by creation of an object of type T). Allocate would create an object of type T, create a pointer to an object of type T, and then assign a value to the pointer so that it points to the object just created. Hence, these are clearly not characterizing operations on Pointer to type T.

3.4 Object as a Generator

Is Object a type in the CLIDT sense? No, it is a Generator (like Pointer). Specifically, If one supplies a CLIDT type T as a parameter to the generator Object, one obtains the generated type Object of type T. For example, if one supplies the type Integer to generator Object, one obtains the generated type Object of type Integer. If one supplies the type Generated Record of type R (itself generated by supplying the field types of R to the generator Record) to the generator Object, one obtains the generated type Object of type Record of type R.

3.5 Other Kinds of Objects

UID's are only one of several common mechanisms for identifying and distinguishing objects. Others are:

(1) Identification by application specific key, e.g., part number, employee number, purchase order number, etc. As these examples suggest, such ID's are often assigned as arbitrarily as UID's but they are assigned by applications, not by the system itself. Hence, they have some significance, even if only as ID's, outside of any given automated system in which they are in use. In CLIDT, objects identified in this way, correspond to entries in Tables; the identifier is the key. (If the entire value of the object is its identifier, then they correspond to elements of CLIDT Sets.)

(2) Identification by ordinal position. In this case, the objects correspond to elements of a CLIDT Array; the identifiers are the values of the Index.

(3) Identification by naming the object itself. Here the name is typically assigned by a programmer rather than by either a user (Table key) or a system (UID). This is the familiar case of a program variable. It is not commonly viewed as an object to be exchanged. Programmers generally assume that they will output or input values of variables but not the variables themselves. But the advance of object-oriented technology, especially the idea of a "persistent" object, may require this view to be modified. Hence, a reasonable variation of Object of type T would be Named Object of type T. For a given type T, e.g., Integer, Named Object of type T would generate an object type such that each object would be associated with, and uniquely identified by, a user-assigned name rather than a system-assigned UID. Although such names often have mnemonic significance to those who assign them, this is NOT a necessary characteristic of the type.

4. Response to N221

The above discussion suggests answers to some questions raised by Yellin in N221.

4.1 Assignment to a Variable

Yellin points out in N221, 1.3 that:

> [i]n general, the characterizing operations [adopted by CLIDT] do not have any method for specifying the assignment of a constant value to a variable of ANY datatype, nor for constructing a new instance of a datatype (e.g., creating a new record or table).

From the foregoing discussion, it seems clear that assignment of a value or creation of a new instance of a value are indeed necessary but that they are characterizing operations of the generator Object, rather than of types like Integer or generators like Record and Table. However, these operations can be (and in CLIDT WD 4 are) defined for elements of Table, Array, and Set, i.e., for generators that define collections of objects. For example, the operation Insert acts as both a create and an assignment operation for generator Table. Similarly, Replace performs these operations for Array; however, CLIDT currently assumes that all the elements of an array are defined when the array itself is initially defined so that one is not allowed to "create" them one by one. List combines these functions in the Append operation. The operations are not defined explicitly for Set; One could define a "create" as a combination of Setof (which creates a one-element set) and a Union. And so on.

4.2 How Many Characterizing Operations for a Datatype?

Yellin says in N221, 1.3, that there are "inconsistencies in the level of detail provided by characterizing operations for different datatypes." For instance, in N221, 2.3, he says that:

> [p]art of the reason that a complete set of characterizing operations is desirable is that it would allow one to precisely define new derived datatypes. ... [CLIDT WD 4] is deficient in this respect. So, for instance, the definition of stack is derived from the list datatype. However, the definition of its characterizing operation push uses a list operation (adding an element to the front of a list) which was never defined as a characterizing operation.

On the other hand, he objects in N221, 1.3, that "8 characterizing operations are presented for the set CLIDT (many of which are superfluous)."

As pointed out above, even with logically superfluous operations defined for Set, there is no explicit "create" operation, so that one needs to define it using two of the existing operations. Similarly, one COULD define an operation to add an element to the front of a List (which Yellin argues is missing) in terms of the existing operations Append and Head, by the ridiculous process of transferring elements from the front to the back of the list by a succession of Head, Append operation pairs.

It could be argued that since characterizing operations exist (as Yellin plainly recognizes and

states) to define the properties of a type and not to specify an efficient implementation of operations for the given type, it doesn't really matter whether certain operations have to be defined in an inefficient manner. However, my own feeling is that the set of characterizing operations for type T should be "convenient" in the sense that they enable one to think about the properties and manipulations of T in a convenient and natural manner. This consideration is more important than choosing a set of operations that is logically minimal as long as no inconsistency is introduced. So, I would be inclined to add a "create" operation that adds a new element to a set, and an operation that puts a new element at the head of a list. But neither is logically necessary. The addition of an operation that adds an element to the head of a list could be justified by CLIDT Resolved Issue 3, since as Yellin points out, it is used in the definition of Stack.

4.3 Essential Operations on a Type

In N221, 1.3, Yellin draws a conclusion from his reading of CLIDT WD 4 that I think is incorrect. Having come to the entirely correct conclusion that "the fundamental operations allow one to precisely describe the values that a variable of a particular datatype can have in a constructive fashion," he concludes that "there may be an operation which is ESSENTIAL to the datatype, but is not presented in the datatype description, since it is not needed to distinguish this datatype from another."

The characterizing operations define the value space of a type T by specifying what one can do to values of the type and hence what properties the value space must have to support these operations. An operation on T that is not one of the characterizing operations, or derivable from them, would therefore have to be an operation that did not depend on the properties of the value space of the type. Such an operation would not BE an essential operation of the type by any reasonable definition of "essential." Of course, in a given application, a type T might require operations beyond those specified in CLIDT. But such operations would necessarily depend on additional, application-specific properties of the type that went beyond the generic properties specified in CLIDT.