Date: Thu, 30 May 91 16:01:09 -0400
From: magnet@cs.UMD.EDU (Magnet Group)
Subject: Language Independent Datatypes - Outstanding Issues

Outstanding Issues on Language Independent Datatypes
Comments on SC22/WG11 N233 (X3T2/91-109)

by
Ed Greengrass

This is my response to (some of) the outstanding issues in JTC1/SC22/WG11 N233 (X3T2/91-109), "Language-Independent Datatypes, Working Draft #5" (WD 5, for short). Each issue is followed by a short answer (e.g., "yes" or "no") followed by a rationale for the answer. I will follow the practice of referring to the proposed standard under discussion as Common Language-Independent Data Types (CLIDT for short).

****
Issue 1. Should there be fewer distinguished datatypes in clause 7?
****

Answer. No.

Rationale: A comment under issue 1 asks about the relationship of "distinguish" to "defined-types". The relationship is that they are completely different. Datatypes appear in the CLIDT because they are conceptual (as distinct from application-specific) datatypes of use in data/information exchange among programs. They are distinguished if end-users, e.g., programmers and users of these programs, find it useful to distinguish them. On the other hand, type T2 may be defined in terms of T1 for reasons of clarity and economy, e.g., because there is substantial semantic overlap between T2 and T1. It may be quite arbitrary whether T2 is defined in terms of T1 or vice versa. In any case, defining T2 in terms of T1 carries NO implication that T1 is more basic or more important or more generic than T2.

I suspect that the desire to reduce the number of datatypes in clause 7 and the use of the word "primitive" in the title of clause 7.1 both arise from the idea that the clause 7 types and generators are all you REALLY need and that the defined types and generators in Annexes B and C, although specified as "normative" in WD 5, are merely in the "nice-to-have" category. It is this view that I want to combat. Another possible implication is that the number of "primitives" is fixed but that the number of defined types is extensible (just as the number of axioms in Euclidean geometry is fixed but the number of derivable theorems is endless). Again, this is a wrong view of Annexes B and C. The number of application-specific defined types is indeed endless but the number of defined generic types in Annexes B and C is not.

Hence, the important question to be resolved here is whether the various pairs of types contrasted under Issue 1 should be distinguished. A secondary question is whether one of the two should be relegated to the Annexes. I conclude that:

(a) Enumerated SHOULD be distinguished from State since the values of Enumerated are ordered and the values of State are not, a most important conceptual distinction. It is a matter of taste whether Enumerated is defined in terms of State or not. Note that for purposes of exchange representation, a single type family (the ordered type Enumerated) is sufficient since a State type can always be represented by an Enumerated type but not vice versa. Conceptually, the distinction is important.

(b) Bit SHOULD be distinguished from Boolean since they are thought of quite distinctly by users. The fact that they are mathematically isomorphic means that one can certainly define one in terms of the other, and probably should. Which one stays in clause 7 and which one is moved to Annex B is arbitrary. Reasons of symmetry would suggest that they both should remain in clause 7.

(d) Set SHOULD be distinguished from Bag because the uniqueness constraint on elements of a set vs. THE NON-UNIQUENESS CONSTRAINT ON ELEMENTS OF A BAG is an important conceptual distinction to at least one community of data exchange users, database users. For example, two distinct measurements in an experiment or two distinct but identical responses to a poll should be preserved as distinct elements: there is a big difference between ten voters with similar opinions and one voter accidentally surveyed ten times! On the other hand, two distinct personnel records for the same individual should not be preserved just because the person was inadvertently required to fill out an application form twice! Nor are these distinctions only of interest to database folks: the advent of object-oriented technology is blurring the distinction between program objects and database objects, the latter merely being "persistent" program objects. The distinction between sets and bags is so important, e.g., relational theory is set-theoretic, not bag-theoretic, that it is probably more misleading than useful to define one in terms of the other: Set and Bag both belong in clause 7.

(e) Array SHOULD be distinguished from List. Moreover, it should NOT be defined in terms of List, i.e., it should not be viewed as a variant of List. Some of the fundamental differences between Array and List are: One can insert elements into a list but only replace the value of an element in an array. Correspondingly, Array is fixed length and List is not. One can have an empty list but not an empty array (only an array with blank- or null- or zero- valued elements). (The above points are copied directly from WG11/N211, X3T2/90-314.) In addition, as Meek has pointed out, Arrays may be multi-dimensional, Lists may not: a list of lists is not the same thing conceptually as a multi-dimensional Array. Hence, List and Array both belong in clause 7.

(f) Array is NOT a special kind of Table because Table, like List, is variable length and can be empty. Furthermore, values within the domain of the Table's key may be ABSENT from the Table, i.e., it is not necessary for a Table to have a row (in WD 5 terms, a value-pair) for every value in the domain of its key. Key values may ALSO be associated with null attribute values. The meanings of these two situations are NOT the same. See issue 4 for a further discussion of Table.

In contrast, an Array is fixed length in the sense that there is no characterizing operation (insert or add) that can change the length (number of elements) of an array dynamically. An Array contains an element for each value in the range of its index; arrays cannot have missing or absent elements. Hence, an Array cannot be empty although it can have blank- or null- or zero-valued elements. Furthermore, the index of an Array is "ordered", e.g., it should be possible to do index arithmetic. By contrast, the key of a Table need not be ordered.

Hence, Table, List and Array should all be distinguished and all belong in clause 7.

(f2) As I pointed out in WG11/N232 (X3T2/91-072), Arrays, Tables, Sets, and objects of type Pointer to T are all collections of Objects, distinguished by how they are identified and hence by the mapping function that maps from an identifier value to the object identified. However, the characteristics of these mapping functions (which are characterizing operations of the generators involved) are so significantly different that it does not seem useful to view them as variants of a common Map type.

****

Issue 3. Is the Character String type primitive?

****

Answer. No.

Rationale: In SC22/WG11 N211 (X3T2/90-314), I argued that Character String was not primitive but should be defined as List of Character. Then, Meek carried matters a bit farther by pointing out that Character String also has array-like operations. Hence, in SC22/WG11 N244, I concluded that there were essentially two kinds of Character String: Fixed Format based on Array of Character and Free Format based on List of Character. In neither case, is Character String primitive. The list-like operations (e.g., character insertion, substring recognition or extraction by va1ue, concatenation, deconcatenation, etc), are most naturally defined (and thought of) on List of Character. Similarly, the Array-like operations (e.g., substring recognition or extraction by ordinal position, etc) are most naturally defined (and thought of) on Array of Character.

****

Issue 4. Should Table be redefined?

****

Answer. No, but its definition should be clarified.

Rationale: As noted under issue 1(f), values within the domain of the Table's key may be ABSENT from the Table, i.e., it is not necessary for a Table to have a row (in WD 5 terms, a value-pair) for every value in the domain of its key. Key values may ALSO be associated with null attribute values. The meanings of these two situations are NOT the same.

The distinction between an absent key value and a key value with a null attribute value goes to the heart of an important database issue: is the table "complete"? Does the absence of an element for key value K1 mean that K1 is known to have a null attribute value or does it mean that nothing is known about attributes of K1. In other words, can we assume that all existing K1, i.e., all K1 in the universe of discourse, are in the table if they have non-null values? The answer to this question depends on the application domain. Hence, CLIDT must remain neutral; this can only be accomplished by defining Table so that the definition distinguishes an absent key value K1 from a a table element with a key value K1 and a null attribute value.

A Table with "multiple keys, analogous to arrays with multiple indices" is quite reasonable and corresponds to the common database concept of a composite key, i.e., a key composed of two or more datatypes. Each component of the composite key would be a "coordinate" of the multi-dimensional Table; the uniqueness constraint would now apply to values of the entire composite key rather than to its components separately.

****

Issue 6. Are Null and Undefined the same datatype or two different datatypes? Is either one a datatype at all?

****

Answer. They are both datatypes. They are NOT the same datatype.

Rationale: I have discussed this issue in a whole series of papers, notably SC22/WG11 N211 (X3T2/90-314) and SC22/WG11 N224 (X3T2/91-070) as well as private communications with Barkmeyer and Yellin. I therefore copy text from those papers into this rationale, suitably condensed and edited to remove inconsistencies. First, however, let me stress that many types have values that can be confused with Null or even used to REPRESENT Null. Lists, Tables, Sets, etc, can be empty, Integers can have the value zero, Characters can have the value blank, and so on. However, none of these is conceptually equivalent to Choice (T, Null) for type T. Barkmeyer has an excellent discussion of these various cases in SC22/WG11 N216.

Null and Undefined are data VALUES, not states or statuses of a variable, for reasons that I discuss below. (A variable can also have an Undefined STATUS but that is something completely different, as I discuss below.) One COULD attach Null and Undefined as values to every CLIDT type, i.e., instead of Choice (T, Null), one could specify that every type T has a special Null value. This would be very messy because the normal characterizing operations of type T would not apply to its Null value and one would have to add a new Null() predicate operation to every type T. It is surely much cleaner to define Null as a separate type.

Choice of a Null type as an alternative to type T should not be confused conceptually with an "empty" variant though it might be REPRESENTED that way. For example, it has been argued that:

```
record of (
   a: integer
   b: choice of (
      integer,
      real,
      null
   )
)
```

is conceptually equivalent to :

```
record of (          record of (          record of (
   a: integer,          a: integer,          a: integer,
   b: integer           b: real           )
)                  )
```

No! This view blurs important distinctions. There are at least two important distinct cases:

Case 1 (the variant record case):

We are defining variations on the SAME record. For example, below I discuss the case of a Personnel record containing a Name of Spouse field. In that case, choosing type Null for the Name of Spouse field means that the person is unmarried and hence doesn't have a spouse. This is an important piece of semantic information. Whether it is represented by a null value or by absence of the field altogether is purely a representation issue. Conceptually, the field has type Null. If the person is married but the spouse's name has not been supplied, then the type of the field would be undefined (or Unknown).

Case 2 (the multiple record type case):

We are defining two or more DIFFERENT types of record which may appear in the same "position". For example, in the above example, the integer "a" might be a record type field so that each

value of "a" corresponds to a different record. In that case, it might be quite reasonable to say that for one record type, the field "b" is absent because "b" might have no meaning for that type of record. But of course, in that case, it would be more correct conceptually to write:

```
choice of (
record of (
   a: integer,
   b: integer
),
record of (
   a: integer,
   b: real
),
record of (
   a: integer,
)
```

to express the idea that three DIFFERENT record types are involved. Alternatively, one might write:

```
record of (
   a: integer,    /* record type */
   choice of (
   body_of_record_type_1,
   body_of_record_type_2,
   body_of_record_type_3
   )
```

where, in the present example, the body is trivial: b: integer, b: real, or b: null.

One can certainly define Null as a (very important) special case of the type State but you won't gain much semantic economy by doing so and Null deserves the prominence of a place in clause 7. In any case, Null definitely should be a distinguished type in CLIDT. Undefined (I prefer to call it Unknown) can reasonably be defined as a generic State type with a set of generic values:

meaningless, unknown, non-existent, not-as-yet-defined,

that has broad (but not exhaustive) generic applicability. Hence, Unknown is a generic type that belongs in CLIDT although the definition of application-specific extensions to Unknown or application-specific alternate value sets for Unknown belongs to other standards.

What follows is my argument (from N211 and N224) on why Null and Undefined (better Unknown) should be viewed as distinct and important types (whether defined in terms of State or not), not program statuses. As usual, I argue by example:

Consider a personnel record with a field Character String for Name of Spouse. What should that field be for an unmarried person? Answer: an instance of type Null, meaning that there is no spouse. Note that this is n o t a null character string but an instance of the distinct type Null (which has only one value, "null", in its domain).

Now, suppose that the spouse DOES exist but we don't know his/her name because the employee specified his marital status as "married" but accidentally left the "name of spouse" field blank on his entry form. In that case, we choose to represent "Name of Spouse" as a value of type

Undefined. This means that we know the spouse exists but we don't know his/her name. The spouse's name "value" is undefined in the sense that it hasn't been specified to us.

Hence, Null is a datatype whose value EXPLICITLY denotes non-existence. Undefined (or as I prefer, Unknown) EXPLICITLY denotes that the value DOES exist but is not currently known.

Note that there is an important distinction between Null and Undefined. Null is a "normal" case, i.e., it is perfectly acceptable for an employee to be unmarried. Undefined is a "bad" case from the application point of view. We lack data that the designer of the application apparently thought would be of benefit (unless, he was just being nosy)! However, this omission does not invalidate the database and shouldn't cause application programs or DBMS's to raise a program "exception". In fact, we specify an Undefined type precisely so that we can handle missing data without "blowing up".

Undefined is sometimes regarded as the status of a variable, corresponding to the operational programming concept of a variable that has been created but to which no value has yet been assigned. However, as the above example illustrates, significant situations exist in which a variable HAS a value with the semantics, "This variable has a value but the value is currently unknown". In different application domains, this value would have more specific meanings, such as, "X refuses to supply the value," "X failed to supply the value," "The value has not yet been obtained but will be," "The value is unobtainable," etc. A reasonable name for this value domain is Unknown. Following Barkmeyer, one might define Unknown as State (refuses_to_supply, failed_to_supply, not_yet_obtained, unobtainable).

Barkmeyer has argued that Undefined is not a datatype because it has no characterizing operations. Specifically, he says that "[a] comparison between two objects which are undefined or "partially undefined" is itself "undefined"." This is quite true with regard to a variable whose value has an undefined status in the conventional programming sense. However, a comparison between two objects of type Unknown IS defined. For instance, in my example of a record with spouse's name unknown, equality of records would mean, "In both these cases, the individual in question has not supplied the name of his spouse." If the records represented successive points in time, the meaning would be, "The individual has STILL not supplied the name." If records of two different individuals were compared, a comparison of the "name of spouse" fields would mean, "These individuals have both not supplied the names of their spouses." It could be quite reasonable to ask for the names of all employees who have not supplied the names of their spouses. Needless to say, equality of unknown spouse names does NOT mean that the names are equal, but rather that they are equally unknown. Instead of Choice (List of Character, Null) where choosing Null means that the individual is unmarried and hence there is no spouse name, one has Choice (List of Character, Unknown) where choosing Unknown means that there is a spouse but the name of the spouse is currently unknown.

It should be stressed that a variable can pass from a value of its type to a value of type Null to a value of type Unknown. For example, if an employee gets divorced, his "name of spouse" becomes Null. If he then remarries but fails to supply the name of his new spouse, the "name of spouse" becomes Unknown. Choice (List of Character, Null, Unknown) expresses this without saying anything about how one gets from one field type to another. One program might delete the old record (with "name of spouse" = List of Character) and create a new record (with "name of spouse" = type Null) in its place. Another program might dynamically delete the "name of spouse" field and dynamically add a field of type Null in its place. Such operational details are outside the realm of CLIDT. Choice only specifies the legal record alternatives, not how one gets from one to another.

How does the Unknown type, as discussed above, relate to the Undefined status as used by Barkmeyer in N216? One can certainly create a personnel record, assign values to its fields but NOT to the field "name of spouse". Does this mean that "name of spouse" is Undefined in the N216 sense? Not necessarily. To decide, it is necessary to examine the state and logic of the program involved. For example, suppose program P creates a personnel record and immediately assigns values to all

its fields. If P is stopped or interrupted before the assignment to "name of spouse" can be executed, then "name of spouse" is indeed Undefined in the N216 sense. However, suppose P creates personnel records by obtaining its data from a heterogeneous collection of databases. Further, suppose that all of these databases record marital status but some of them do not record data about spouses. If P tries to create a record about married employee X and finds that data about X is available only in a database that does not support spouse data, then AT THAT POINT "name of spouse" passes from Undefined to Unknown; P should express this by setting "name of spouse" to type Unknown, value = unavailable.

The distinction is not trivial. The fact that "name of spouse" is Undefined only tells us something about P, i.e., that P has not gotten around to assigning a value to "name of spouse" for X. The fact that "name of spouse" for X is Unknown tells us something EXTERNAL to P, something about the "real world" or at any rate about the environment in which P operates. Hence, Unknown is a value of a datatype. Undefined is a status of a variable corresponding to the current state (the logical program counter) of program P. It follows that to compare two Undefined variables only tells us something about P (if that). Comparing two variables of type Unknown tells us something meaningful about the world outside of P itself.

****

Issue 8. What is the anticipated use of outward mappings?

****

Answer: Outward mappings will be used in all the same ways and may encounter all the same problems as inward mappings. The governing consideration is that in any given service exchange (a CLIPCM procedure call being one example), inward and outward mappings must be inverses whether or not such mappings have been specified by the relevant language communities. If they are not specified by the language community, they may be specified by a service community, e.g., CLIPCM/RPC, or by a given application community. These issues are discussed at some length in the Barkmeyer-Greengrass dialogue, SC22/WG11 N246.

Rationale: The idea that outward mappings are somehow less basic than inward mappings is wrong. In the ideal case, which will actually often occur, there is a unique one-one mapping between CLIDT type D and the internal type E of a given language FRIML. Hence, the outward mapping M: E => D and the inward mapping M: D => E are inverses defined by the FRIML language community (Barkmeyer calls these "semantic mappings") and happily used by services such as CLIPCM and RPC and by all application communities that write in FRIML. For example, consider a call by a FRIML program to an advertised procedure SNARK (written in FRIML) whose IDN specifies an inout parameter of CLIDT type D. The call will specify an parameter of type E which will be mapped into type D for exchange, then mapped into type E inside SNARK, then mapped back into D on exit from SNARK, then mapped back into E when control returns to the calling program. Everyone is happy!

Problems arise when the CLIDT to FRIML mapping is not one-one. In general, we may assume that CLIDT has a type system that is at least as powerful and probably more powerful than FRIML. This leads to such possibilities as:

Case 1: CLIDT being more powerful than FRIML, M'(D1) => E and M'(D2) => E. The inward mapping M is still unique. The FRIML community can define a unique semantic outward mapping, e.g., M(E) => D2, the choice being arbitrary if there is no better reason. Service communities that want to use both D1 and D2 in their IDN's must define non-unique outward mappings (which Barkmeyer and Schaffert call "inverse inward mappings"). Note that in this case there is a loss of meaning in the inward mapping, i.e., some distinction(s) between D1 and D2 is lost. This may be acceptable, e.g., because the distinction is unimportant to the relevant application community or because the lost semantics is preserved

in encoding conventions specified outside of CLIDT, e.g., in the CLIPCM marshalling/unmarshalling standards or in the standards of the application community.

Case 2: M'(D1) => E1 and M'(D2) => E2. However, the service community chooses, wisely or not, to support D1 but NOT D2. Hence, service calls from FRIML programs must define M(E1) => D1 and M(E2) => D1. This is a unique outward mapping but different from the semantic outward mapping. (More precisely, it is a weaker semantic mapping, the best that can be done with the given constraints.) FRIML service programs will not use D2 as an operand in their IDN prototypes. They may choose not to use E2 in their FRIML prototypes either to remove ambiguity.

Case 3: M'(D1) => E1 in some applications and M'(D1) => E2 in other applications. In other words, in some cases it is the E1-like characteristics of D1 that are of interest, in other cases it is the E2-like characteristics that are of interest. Now, the outward mapping is unique but the inward mapping is not. Again, there is a loss of meaning in the inward mapping which may not matter to the relevant application or service community or may be compensated by encoding conventions outside of CLIDT.

From the above considerations, I conclude that many important mappings may use CLIDT, i.e., may be defined to and from CLIDT types, but may depart from strict compliance because they may require pragmata and/or encoding conventions to account for departures (inward, outward or both) from a perfect mapping. Also these mappings may be defined by language communities, service communities, application communities, or joint working groups drawn from two or more of these communities, depending on the specific circumstances.

****

Issue 11. What is the proper model of Pointer datatypes?

****

Answer. Pointer is a Generator. It has the value space of Unique Identifier (UID) but is NOT "a name for the primitive datatype whose values are the "unique identifiers" of the objects pointed to."

Rationale: I have discussed Objects and Pointers at length in SC22/WG11 N232 (X3T2/91-072) and further in SC22/WG11 N247. In the former, I explain why Pointer is a generator. I also explain why pointers point not to values but to INSTANCES of values (or, as I call them, Objects). Specifically, two distinct Pointer values can point to two distinct instances of the same value, e.g., two integers having the same value five. Distinct values of the same type are distinguished by Unique Identifiers (UID's). In the April X3T2 meeting in Gaithersburg, it was pointed out that a collection of UID,value pairs is a Table in the CLIDT sense and can usefully be viewed that way. Hence, in N247, I recognize that what I called Object of type T in N232 can be redefined as Table of (T, UID) where UID is the Table key.

In N247, I also explain the difference between the primitive type UID and the generator Pointer. Consider an Integer object pointing to another object in a linked list. The first object could be defined as: Record of (intval:Integer, nextval:Pointer). The second object could be defined as an element in Table of (T, UID). In the Table definition, I call the type of the key UID. In the Record definition, I call the type of field nextval Pointer. Since the value space of Pointer to type T consists of all UID's identifying instances of type T, what's the difference between Pointer and UID? The difference is in the characterizing operations. The nextval field is a Pointer because it makes sense to apply the dereference operation to it to get the object pointed to, the next element in the linked list. However, in the Table definition, the UID functions as a key, identifying the object in which it occurs, not pointing to any other object. Hence, dereference would make no sense and the type is NOT pointer even though the value space may be the same. In short, UID is a primitive, but Pointer is a generator because dereference only applies to Pointer, not to UID.