3/14/91

To: CLIDT, CLIPCM Folks
From: Ed Greengrass
Re: Types, Subsetting, Levels of Concern, Part 2

This is a continuation of my previous comments on the subject. I'll just keep running on at the mouth until I run out of words (I almost said "run out of gas" but that would have mixed metaphors!). Rereading my previous comments, they seem to be coming out as a tutorial on Ed's ideas but perhaps they'll depart a bit from Ed as I continue.

Ed has expressed the view (in a private communication) that State must be supported at the exchange level but Enumerated need not be since it is "adequately supported, like Ordinal, by datatype Integer." In terms of my previous discussion of Ed's levels of concern, it seems clear that he means that Enumerated can be encoded at the exchange level as an integer. However, the integer value must be tagged, i.e., its type must be labeled. Interpretation of this label is NOT outside the realm of CLIPCM. Rather, the label(s) must be interpreted in mapping from the exchange to the user level of CLIPCM. Hence, I would say that Enumerated (and State, and indeed all CLIPCM datatypes) should be supported at the user level of CLIPCM (and other exchange standards at the same or higher application layer). I suppose the key question here is whether it is meaningful to talk about the "user level of CLIPCM". In other words, when a service advertises a procedure and a user invokes that procedure, is their level of discourse, which should certainly include Enumerated, etc, part of CLIPCM or above it? I believe it is part of CLIPCM.

Let me elaborate on this by discussing the difference between List and Array and the practical effect of the difference on procedure calls. A crucial difference, quoting from my X3T2/90-314, is that "one can insert elements into a list but only replace the value of an element in an array." Now, suppose that program A calls procedure P and the Ith input operand is an inout value of a List type. P should be able to insert an element in this list. Let P insert an element between the jth and (j+1)th elements on the original list. After P returns to A, A should be able to read this new element and the element following. If A treated the list as an array, saved off the index value j+1 before the procedure call, and tried to read the (j+1)th element after the call, it would get the new element and not the element it expected. It would "think" that the old value of element j+1 had been replaced by P which would be incorrect.

Superficially, it might appear that this difference is "only" an operational difference and has nothing to do with exchange of datatype values, but this is not so. Consider two cases. First, suppose the operand in question is represented in traditional array fashion, i.e., as a fixed list of equal length elements so that the integer index j (assuming the first element is one) can be used to access the jth element. Plainly, with this representation, insertion of an element by P is easy (assuming that P can get more memory as needed) but such an insertion will mess up A's indexing as already discussed.

Now, assume that the array is represented associatively, i.e., each element in the array is accompanied by its index value. (In effect, it becomes a Table with ordered keys.) Now, A will have no difficulty accessing element J no matter what its physical position, but P will be unable to insert an element logically "between" the Jth and (j+1)th elements because there is no index value between j and j+1.

Could on devise a way, with sufficient ingenuity, to have it both ways? Not really. If P is allowed to do insertions, then they have to be done in such away that they don't "look like" insertions to A which means that P and A would have to interpret the operand value inconsistently. That does not seem to be exchange of datatype values as I would use the phrase. For example, P would have to look at logical ordering links that A ignored and A would have to look at index values that P ignored. Some exchange!

However, Barkmeyer has argued (I've just reread his argument) that List is more general than Array.

Hence, one can define an Array as "a List of values, with the subscript mapping function attached." This is along the lines of my first example above, with the index value (the "mapping function") "attached". This would not be at all satisfactory at the user level (a user doesn't want to bother thinking of his array as a special kind of list) but arrays can certainly be exchanged that way. Barkmeyer says that "[i]n CLID terms, that is a NEW List datatype." In other words, you LABEL a given List type as an Array. At either end, this exchange value has to be mapped into a "real" array which is what the user is talking about. If he sees a list, he will be tempted to do list-like things (like insert) to it, which is a no-no as discussed above. Alternatively, if he is thinking of an Array, seeing a List will confuse him.

Again, the question is whether CLIPCM is concerned with the user view of procedure calls or only with the exchange protocol. I suppose that one could argue that if an application program user writing in L1 calls a procedure that requires an Array-valued operand, the array is mapped into a CLIDT List, one of the types "offered" by the CLIPCM model. Hence, the model itself doesn't need to know about Array though the mapper does. But this seems a bit disingenuous. I think that CLIPCM should "know" about Array, if only to recommend ways of "encoding" it for CLIPCM exchange purposes, i.e., at the CLIPCM exchange level.

Further, the advertiser of procedure P is in a different position than the caller of P in program A. The caller may call P using the datatypes of "his" language, L1. But if the P advertiser wants his procedure to be available to a multi-lingual community, he won't advertise it in the language, say L2, in which he happened to write P. He will advertise it, at least the datatypes of its operands, in generic CLIDT terms. One of those operands could easily be an Array. Hence, if CLIPCM is prepared to call this advertised procedure, it should be ready to "talk" full CLIDT, even if it exchanges the array using a list "encoding".

I have not said any thing about Ed's third level of concern, the "language" level. Actually, my "user" level seems to be equivalent to Ed's "language" level since I have only been dealing with what the user can express. Implicitly, I've assumed that his language L1 is satisfactory for that purpose. What if it isn't? For example, what if his datatype set is weaker than the CLIDT set required by the procedures he wants to call? Even if that isn't so, there will certainly be things he "knows" about the datatypes that CLIDT doesn't allow him to express. How does he express (and exchange) this additional knowledge. Presumably, in the traditional ways: by use of mnemonic names for variables that tell humans more than they tell programs, by human-to-human communication of knowledge that is then buried in the code of the sending and receiving programs, by conventions for encoding this knowledge in the values he exchanges via CLIPCM (thus, a primitive CLIDT value may encode knowledge according to rules understood by the communicating programs but NOT by CLIDT, etc. Eventually, he may be able to use a conceptual schema exchange standard for this purpose.

Best Wishes
Ed Greengrass