From: magnet@cs.UMD.EDU (Magnet Group)
Message-Id: <9104161934.AA29647@tove.cs.UMD.EDU>
To: willemw@ace.nl
Subject: CLIDT - Two Dialogues and Four Papers on Distinct Datatypes

From: Ed Greengrass
To: WG11 CLIDT-ers
Subject 1: Bit vs. Booolean - A Greengrass/Barkmeyer Dialogue
Subject 2: Date-and-Time - Another Greengrass/Barkmeyer Dialogue
Subject 3: Distinct Datatypes

This paper requires an historical note. It started with Brian Meek's provocative "two quids worth " paper to SC22/WG11 on "distinct datatypes". Meek, a professed "maximalist" where CLIDT is concerned, called for other maximalists to rally round. As I was preparing to do so, I received a message from Ed Barkmeyer, responding to Brian. Ed wrote (in part):

> [T]here is an open question: when two types have
> isomorphic value spaces, and one has a subset of
> the operations of the other, are they distinct
> types in some primitive way, or is it sufficient
> to describe the second as derived from the first
> by eliminating operations? This provides a ground
> for debating whether:

a. Bit and Boolean are both primitive datatypes.
b. State and Enumerated are both primitive datatypes.
c. Ordinal and Integer are both primitive datatypes.
d. List, Bag, and Set are distinct generators.
e. Table is really distinct from Bag of Record(key, element).
f. CharacterString is distinct from List of Character.
g. BitString is distinct from List of Bit.
h. Handle/ObjectId is something different from Pointer to Private.

I immediately responded by writing a series of four short papers dealing with all the above cases (except Ordinal vs. Integer), discussing my views in terms of Meek's and Barkmeyer's. This set of papers is included as Part 3 of this lengthy message. (Barkmeyer's response to Meek was addressed only to Meek, Schaffert, Yellin, Rabin and myself so I used a similar address list originally.)

Barkmeyer responded to my papers by disagreeing privately with my view on Bit vs. Boolean. The result was a dialogue, less momentous than those of Plato no doubt, but still worthy of consideration. At Barkmeyer's suggestion, this dialogue is included as Part 1 of this message.

We also had a shorter dialogue on the subject of Date-and-Time which I include as Part 2 of this message.

Bit vs. Booolean - A Greengrass/Barkmeyer Dialogue

From: Ed Greengrass
To: CLID-ers
Subject: Meek's Two-Quids Worth and Barkmeyers Response - the Sequel

Bit vs. Boolean

I haven't read Meek's paper on the subject but I would distinguish Bit from Boolean as he does. My argument is that their spaces are NOT isomorphic, defining isomorphism as I did in the previous installment. The reason is that while Bit and Boolean are both two-valued, knowing that a given variable has a bit-value of "one" doesn't tell you whether that value corresponds to TRUE or FALSE. On the other hand, knowing that two variables have the boolean values TRUE doesn't tell you whether their product is "one" (TRUE = 1) or "zero" (TRUE = 0). Of course, it is also plain from CLIDT WD 4 that Bit and Boolean have different characterizing operations corresponding to these distinct value spaces.

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Re: Datatypes, Two Quids Worth, The Sequel

A quick note.

It is possible, from the characterizing operations (OR and NOT) to derive the operation XOR on the Boolean datatype, and to discover that Boolean is a group under XOR. It is also possible to derive the operation AND and to demonstrate that Boolean is a semi-group under AND, and to demonstrate that the distributive laws hold: (A XOR B) AND C = (A AND C) XOR (B AND C). It is not, to my knowledge, possible to do this with other operators on the Boolean type. So Boolean is a ring under XOR and AND. And the multiplicative identity (the AND-identity) is TRUE, while the additive identity (the XOR- identity) is FALSE. It follows that the isomorphism must map TRUE into 1 and FALSE into 0, so that the operations are correctly preserved on the Bit space, with the Bit characterizing operations (the two-valued numeric ring).

In other words, your first argument is mathematically incorrect.

-Ed

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Bit vs. Boolean Revisited

Dear Ed,

Bit vs. Boolean

Your mathematical argument that correspondence of Bit and Boolean implies TRUE => 1, FALSE => 0 was fascinating and correct. However, I don't think our arguments actually contradict each other.

The essence of my argument was that if you send me values of type Bit, i.e., ones and zeroes, which I am to interpret as values of type Boolean, you need to send me "something else" as well. The "something else" that I had in mind was either "TRUE => 1, FALSE => 0" or "TRUE => 0, FALSE => 1".

Your argument (as I understand it) essentially says that instead you could send me "XOR => Add, AND => Multiply". (It appears that it is the "AND => Multiply" part that is crucial here. The Bit Multiply truth table is only equivalent to the Boolean AND truth table if TRUE => 1, FALSE => 0; the XOR and bit Add truth tables are symmetric so that True => 1 and TRUE => 0 are equally good.)

I think the reason you regard your argument as contradicting mine is that you regard the AND <=> Multiply truth table equivalence for TRUE => 1 as "given". Indeed, it IS given in CLIDT WD 4. The question is whether the statement, "Interpret Bit as Boolean" implies "Map Multiply into AND" or whether the latter is something you have to send to me along with the ones and zeroes. Suppose you sent me, "0 => TRUE, 1 => FALSE" instead. Would that lead to a logical contradiction? No, not if you had done the reverse mapping, "TRUE => 0, FALSE => 1" (or equivalently, manipulated internal ones and zeroes according to a 0 AND 0 = 1, etc, truth table, rather than according to a Bit arithmetic truth table). In other words, the contradiction only arises if you use the internal "trick" of using Bit arithmetic for Boolean agebra in your program or I do the reverse in mine.

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Re: Bit vs. Boolean Revisited,

The mathematical argument is not whether the truth table is "given", but whether the NAND operator, which has the complementary truth table, would do equally well as a Multiply. I.e. is Boolean a semi-group under NAND? (yes) and do XOR and NAND distribute properly? (I'd have to check.) More importantly, XOR is symmetric, but only False will do for the identity. False XOR x = x for all x; but True XOR True is not = True!

The further argument is that I don't have to tell you XOR is Add, because of the isomorphism: your Bit and my Boolean have equivalent characterizing operations - they do exactly parallel things to the value space. So if that's how I know which datatype it is, it must be the same one.

The issue of how it is represented is beside the point entirely. Whether you represent Bit values as binary 0 and 1 or display-code '0' and '1' or of the datatype identification problem. If what you are saying is that you may wish to represent values of the two datatypes differently, then yes, you want to have Annex B say: type Bit = new Boolean. But that means "Bit is different from Boolean because I say so", which is fine with me.

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Bit vs. Boolean - We'll Get Together Yet!

Dear Ed,

OK, back again. Let me restate the problem a little differently.

Let's suppose that we decide that there is no need to distinguish Bit and Boolean. Hence, in the interests of scrupulous neutrality, we replace them both by a type with the ugly name Two-Valued-Space (TVS) having the neutral values A and B. Now, suppose that I write a procedure P in a language L2 that supports Bit but not Boolean. The first operand of P is an input Boolean variable which I am naturally representing internally as Bit but advertising publicly as TVS. You want to call P from your program AP written in language L1 which supports Boolean but not Bit. So, on your end, Boolean is mapped into TVS. On my end, TVS is mapped into Bit. Will this work? Note that I have said NOTHING about the exchange representation.

It seems to me that according to your argument, it will work only if your internal TRUE becomes my internal 1 and your internal FALSE becomes my internal 0. You can ensure this if you know my inward mapping when you are specifying your outward mapping, but you shouldn't have to know that. (Actually, it will work with the reverse mapping too, but only if I know that the reverse mapping is in effect and write more complicated code to take this into account.)

I may try to clarify matters by advertising my operand 1 as Boolean = New (TVS). Even better, I may advertise the operand as, say, Husband Older-Than-Wife? = Boolean = New (TVS). That tells you (in English, of course) how I plan to use that operand. But it doesn't tell you whether A = TRUE or FALSE. That will have to be specified SOMEWHERE, e.g., in my definition of Boolean.

On the other hand, suppose that Bit and Boolean are distinct primitive types in CLIDT, as at present. Now, P advertises operand 1 as CLIDT type Boolean. You map outwardly from your Boolean to CLIDT Boolean. (This can be either the semantic outward mapping or the inverse [of] inward mapping; it doesn't make much difference here.) I map inwardly from Boolean to BIt. This is the unique (semantic) inward mapping from CLIDT Boolean to L2. L2 will naturally tend to map CLIDT Boolean to Bit and, more specifically, TRUE to 1, for the reasons you have given. Even if the L2 community, for perverted masochistic reasons of its own, specifies a different inward mapping, that doesn't concern you at all. It just means that I will have a harder time coding P because I will have to accommodate the idiosyncratic mapping of the L2 community.

Ed G.

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Re: Bit vs. Boolean - We'll Agree Yet!

I think your problem is solved by getting the complete mapping. Assume the CLI datatype TVS is stated to have two values A and B and the characterizing operations Flim and Flam, defined by tabulation. Then when the Pascal standard identifies its datatype "boolean" with TVS in the inward mapping, it must say that A maps to "true" and B maps to "false", and Flim corresponds to "not" and Flam to "and". Similarly, when PL/I identifies TVS with "Bit" in the inward mapping, it says A maps to 1 and B maps to 0 and Flim(x) = 1-x and Flam(x,y) = x*y. Now if you write in PL/I and advertise your CLI interface as requiring a value of type TVS, you are implicitly claiming you use the values corresponding to A and B consistently with the CLID - PL/I inward mapping. If I write in Pascal and my use of True and False is consistent with the CLID-Pascal mapping, then when I put a Boolean variable in the call to P, whatever value it has will be interpreted properly by P. The remaining requirement is that the provider of CLI services for the CompuWunder 4000 gets the physical mapping of the Pascal, CLID/CLIPC, PL/I representations right. But the mappings clearly tell him what physical mapping is wanted, assuming he knows the representation of "true" and "false" in the Pascal compiler and library.

As the current clause 10 says, the mapping has to tell you what the value space mapping is, not just that the types correspond. This is where you notion of "strained" comes in. If the value mapping takes 3 pages to explain, then the type correspondence is gravely in doubt. If it can be done in 2 sentences, as I just did, then the mapping seems to me to be fairly natural.

The door is still left open for type Bit = new Boolean, because my application, or my language, distinguishes them semantically. And

BitString = new [packed] Array of Boolean

may be an important datatype, because its representation is very different from Array of Boolean. But neither of these has anything to do with whether the primitive types are intrinsically distinct.

-Ed

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Bit vs. Boolean - "By George, I Think You've Got It!"

Dear Ed,

Bingo! You're right and your argument leads to some interesting conclusions. First, we started out talking about when two CLI types could be viewed as isomorphic. Now we seem to have arrived at a rule for isomorphic mappings:

> Given a CLI datatype D and an internal type E in some language FRIML, the mapping is ISOMORPHIC (in a sense "perfect") if one can define characterizing operations on E in terms of the characterizing operations of D such that the mapping from the value space of D to the value space of E is uniquely defined in the sense that equivalent operatiions on E and D give equivalent results only for that mapping.

Plainly, the mappping from TVS to Boolean or Bit in any reasonable language satisfies this definition, which is exactly your point. Of course, instead of defining the CLI type TVS, one could drop Bit from CLIDT by saying that it is isomorphic to Boolean. CLI Boolean would then be isomorphic to Boolean or Bit in any language FRIML.

We can view the mapping from TVS value A to Pascal Boolean "TRUE" as two-stage. First, we map A into the Pascal source value for TRUE, a value that is presumably fixed by the Pasccal standard. Second, we map this Pascal source value into the encoding of TRUE produced by the the Pascal compiler under consideration.

Unfortunately, not all CLI types have the agreeable properties that allow such perfect mappings. Consider the type (more precisely the family of types) State. For concreteness, consider a member of this family: XPROTOCOL (WAIT_FOR_SEND, WAIT_FOR_RECEIVE, IDLE, SENDING, RECEIVING) or the like. Let XPROTOCOL be the type of operand 1 of procedure P. Let P be written in L2. Let it be called by program AP written in L1. Then, if AP sends WAIT_FOR_RECEIVE, we want P to receive WAIT_FOR_RECEIVE.

The only characterizing operation specified for State in CLIDT WD 4 is "equivalence". If this means, as I assume, equality of values within a state type, then it allows AP to tell if two given values of XPROTOCOL are equal and ditto for P. But it doesn't fix the CLI to L1 (or L2) mapping.

When we try to map XPROTOCOL as we did TVS, two difficulties arise:

(1) We would like the mapping to be generic, i.e., to apply to all types that are members of State. An obvious way to do this would be to map from the order in which the state values were specified in the definition of XPROTOCOL (a public definition) to the corresponding definitions in AP and P. But this uses a property of the syntax of the definition, the order of listing of the state values, which is NOT part of the semantics of the type. Indeed, if one of the languages involved was a 2-dimensional visual language, the linear ordering might not even be defined. Moreover, it is quite possible that we might want to map from a given State type to another whose space was a subset of the former. Finally, it requires everyone who defines a version of the given type to list values in the same order (presumably the order in which they were

listed in P's public definition of XPROTOCOL).

(2) We would like the mapping for L1 or L2 to be independent of the particular code of AP or P. This was easy for Boolean to Pascal because the Pascal way of saying TRUE is (I assume) standard. But the way of saying WAIT_FOR_RECEIVE is not so standardized. Again, one could use the ordering of the definition; this means that if WAIT_FOR_RECEIVE is the second value listed for the type, everyone has to list it second, and two will, in effect, "mean" WAIT_FOR_RECEIVE.

The closest we can come to avoiding these difficulties is to require each programmer to either (1) use the exact same name for each value as was used in the CLI advertised definition, or (2) specify a mapping from these advertised names to the names he did use. In other words, in case (2) each programmer must specify what I called stage 1 of the two stage mapping I described above for Boolean. Of course, this will have to be done separately for each program that does not use the publicly advertised names. Why would any programmer want to use different names? Perhaps, because his language arbitrarily limits state names to, e.g., six characters. Or, perhaps, because certain state names are already in use and changing them would involve changing too many programs. In case (1), we have achieved an isomorphic mapping not by any constraint imposed by the characterizing operations but by forcing identity of the value spaces in stage 1. This is not too unreasonable because the value space consists of names that can (and typically do) have mnemonic "real-world" significance.

What about other families? Enumerated presents no problems because the ordering that was introduced arbitrarily in State is part of the semantics of any Enumerated type. Ditto for Scaled. Ditto for Date-and-Time; I would add that though the potential number of members of this family is very large, certain members, e.g., "month, day, year" are very widely used to the point of virtually being standards.

(Incidentally, the view that the ordering operation(s) on an Enumerated type provide the basis for an isomorphic mapping does not contradict the claim I made in an earlier message: that the names of the values themselves have semantic significance so that a mapping from Enumerated to Range of Integer involves a loss of meaning.)

What about Generators? Of course, any Generator could have a State type as a component type which would get us back to the original problem. But, assuming that the component types have isomorphic mappings, what about the generated types? Record provides a situation analogous to State (unordered names) except that the names are attached to field-values instead of BEING the field values. Hence, a mapping (ideally identity) must be specified between the advertised field names of the CLI definition of a Record type and the corresponding internal names in language FRIML.

Ed G.

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Bit vs. Boolean - One Last Gasp

Dear Ed,

In an earlier message, you said,

> [Y]ou [may] want to have Annex B say: type Bit =
> new Boolean. But that means "Bit is different
> from Boolean because I say so"

But in light of our discussion of the isomorphism of Bit and Boolean, it seems to me that that would not be entirely satisfactory. Wouldn't it be necessary to say something like: type Bit = new Boolean with MULTIPLY = new AND and ADD = new XOR? In other words, it is not only the type that is being renamed; the characterizing operations are being renamed too.

You ended your last message by saying, "I believe there are several people who would object strongly to the ordering requirement." Which ordering requirement were you speaking of?

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Re: Bit vs. Boolean - "By George, I Think You've Got It!"

More accurately, we've got it. We should probably extract the right set of paragraphs from our intercommunications and send the result to WG11.

I also agree with your second discussion, requiring that either the state and field names be mapped algorithmically into a programming language, or that the CLID ordering be imposed on the language bindings. I generalized your "identically" into "algorithmically", because a uniform substitution of symbols, or even first-n-characters, is a satisfactory method in most cases, and provision for the programmer to specifically define the value mapping completes the general requirement. There is a requirement for a value mapping to be defined anyway, but for records, it is a field-mapping not a value mapping which is needed. I believe there are several people who would object strongly to the ordering requirement.

-Ed

Part 2
Date-and-Time - Another Greengrass/Barkmeyer Dialogue

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Date-and-Time

Dear Ed,

Your statement that Date-and-Time is an Enumerated type led me to some serious pondering. It seems to me that while it is POSSIBLE to view Date-and-Time that way, it is not a satisfactory viewpoint. I was going to say that it is "strained" but I admit that term is subjective. Let me see if I can state my objections more objectively.

For concreteness, consider one member of the Date-and-Time family, the member each of whose values consists of Year, Month, Day, Hour, Minute. Then a value of this type is "1991, February, 28, 1451" or some equivalent. On the other hand, "1991, February, 29, 1451" is NOT a value of the type. These facts can be determined by inspecting the enumeration. Similarly, one can determine by inspection that "1991, April, 31, 1451", "1991, May, 34, 1451" and "1991, FRIML, 28, 1451" are not values either. However, enumeration does not specify any of the principles underlying such determinations, i.e., that no month has more than 31 days, that April has only 30 days, that there is no month named FRIML, etc. Our calendar (and indeed all other calendars too) may be a trifle erratic but it is not totally without rhyme or reason and an enumeration does not specify its underlying principles. One might figure out these principles by studying the enumeration very carefully but that is not the same thing. Further, deduction from one member of the family would not carry over to other members although the rules would carry over for a given calendar. Finally, one would have to view each enumerated value as composed of substrings (sub-values) corresponding to different units which is not a concept normally associated with Enumerated.

Hence, I believe that Date-and-Time has semantics which are NOT a part of Enumerated, indeed, not a part of any member of the Enumerated family.

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Date-and-Time

Date-and-Time. You are making date-and-time a Record type when you talk about 1991.02.29 not being a value of the type and about carrying between units when a subtraction is performed. In the enumerated type, the successor of 1991.02.28 is 1991.03.01. Predecessor is simply the inverse of successor, and calculating interval is simply counting applications of successor. All of the calendar stuff is very much dependent on representation conventions. 1 May 1991 at 16:04 GMT and the 9th day of Rahim in the year 1358 of the Prophet at 18:04 CPT may well be the same VALUE! What makes Date-and-Time not an acceptable Enumerated datatype is the concept of Extend and Round, which is meaningless on Enumerated datatypes, but happens to be a CONVERSION operation between Date-and-Time datatypes!

-Ed

From: Ed Greengrass
To: Ed Barkmeyer
Subject: Date-and-Time

Dear Ed,

We evidently agree that Date-and-Time is not a pure Enumerated type. But I don't agree that the "calendar stuff" is "very much dependent on representation conventions." A specific calendar notation depends on specific representation conventions. But concepts like "year" and "day" have an astronomical basis. Units like "second" are presumably defined by international convention (does NIST use a Cesium clock to define a standard second or what?). Some international Date-and-Time standards may have a conventional (as opposed to astronomical) basis but that is obviously not the same as a mere computer representation. Therefore these concepts have an existence independent of specific computer representations and are quite meaningful to users of the type. Enumerated abstracts out some of the underlying concepts of Date-and-Time, i.e., discrete ordering, but not all. Since Round and Extend are intended to capture this semantics, we may actually be saying much the same thing.

From: edbark@cme.nist.gov (Ed Barkmeyer)
To: Ed Greengrass
Subject: Re: Date-and-Time, One last Time

Yes. I think we agree. The standard second is defined in terms of the frequency of the Cesium atom, but that's an interval. UTC (Greenwich time) is the international Date-and-Time standard, from which NIST propagates Mountain Standard Time, whence all U.S. time is derived. NIST actually measures time in milliseconds since January 1, 1956, when they activated the first atomic clock in Boulder, although it is rechecked every some number of months with Greenwich observatory time. This is why all the hoopla about adding leap-seconds and suchlike. But the representation of time in "calendar form", i.e. year, month, day, etc., while subject to the UTC standard, is only stated to be "derived from" standard time. Greenwich Observatory Time is propagated as a single Scaled value representing seconds since some time in 1858. This is the basis for Phil Brown's comment (on WD3) that "all astronomers treat time as a single real number." I suspect that the UTC says "derived from" to avoid forcing the calendar representation to be modified when something like a leap-second is stuffed in. So, while calendars are standardized representations, they have nothing directly to do with standard time. The Round and Extend functions, however, involve a concept of "marks" in the standard time value sequence, and the possibility that one can relate a time-value to the last such "mark" which was passed. Those marks have names which are common in calendars. It occurs to me that there may, in fact, be no standard for the term "year", and its relationship to standard time, except for the UTC.

-Ed

Part 3

Distinct Datatypes

From: Ed Greengrass
To: CLID-ers
Subject: Meek's Two-Quids Worth and Barkmeyers Response

This is a first response to those two interesting contributions. I thought that some of the distinctions that Ed raises questions about were already resolved. In fact, in Monterey, I was told that the first part of my N211/X3T2/90-314 paper was not going to be discussed because no one had raised any objections to the CLI distinctions I was attempting to uphold with my answers to the unresolved issues! However, let me have another crack at it. For the record, I am, if not a maximalist, at least a quasi- (crazy?) maximalist.

Enumerated vs. State

I believe that Enumerated and State are distinct primitives that do NOT have isomorphic value spaces. The reason is that there are representations of State which would not support Enumerated because they would not support ordering. This leads to the more general conclusion that for two datatypes to have isomorphic conceptual value spaces, there should be a one-one correspondence between the set of all representations of the one type and the set of all representations of the other.

Character String vs. List of Character

I have hitherto supported List of Character as a CLI derivation of Character String because it seemed to me (as I said in N211) that character string operations were essentially list-like. Meek raises the interesting point that there are array-like operations, too. What he has in mind, I assume, is that one sometimes refers to a string position or substring by numerical index, e.g., ABC(3-5) is the substring consisting of the 3rd, 4th and 5th characters in the string. What I think is involved is two broad categories of string: fixed format and free-form. Array-like operations are only applied to fixed-form strings. List-like operations are usually applied to free-form strings though one CAN sometinmes apply them to fixed form strings too. Fixed format generally implies fixed length as well. Free-form strings are conceptually variable length. When a free form string is fixed length, it is usually an arbitrary length rather than a conceptual property, e.g., someone decides that the maximum length of a "comment" field is 64 characters, although longer comments are quite conceivable.

Hence, I would say that the two categories have somewhat different value spaces corresponding to fixed vs variable length representations. List of Character is quite satisfactory for free-form character strings. Array of Character is usually satisfactory for fixed format character strings.

List, Bag and Set

List is distinct from Bag or Set for the same reason that Enumerated is distinct from State: because ordering is a crucial property of List. Moreover, there are representations of Set that would not support insertion, e.g., Set as a variable-length array would be ordered but still not support insertion.

On the other hand, it might appear that the value spaces of Bag of Type T and Set of Type T really ARE isomorphic. However, even here there are situations where the spaces are distinguishable, e.g., a wholly associative representation where two identical values ARE the same value. Users of relational systems get around this by assigning arbitrary IDs to distinguish otherwise identical rows. In the absence of such assignment, which effectively changes the representation, Sets are representable but not Bags.

In any case, even if the value spaces of Bag and Set ARE viewed as isomorphic, the types should still be distinguished, simply as a way of conveying the information that certain characterizing operations have been employed in generating values of the type, e.g., calling a type a set says, "don't expect duplicate element values", a most significant piece of semantic information.

Handle, Object-ID and Pointer to Private

I've already discussed Objects and Object Id's in my paper WG11/N232 (X3T2/91-072). Here, I'll crib from a message I sent to Ed a few days ago.

Yes, Pointer to Private is Handle. The essence of Private is that there are no known characterizing operations within the data exchange community of interest. Hence, if someone sends you a Pointer to Private, the only thing you can do with it is send it back; only the originator knows the characteristics of the type and hence he is the only one who can do anything else with the object that a given handle identifies. This is exactly the characteristic of Handle.

Pointers and Table keys both contain Object IDs. The distinction between Pointer and Table key is the context and duration to which it applies. Social Security Number (SSN) is an example of a Table key because it is defined externally, e.g., by the U.S. Government, and hence might be of interest to a user; the actual SSN value might be retrieved and examined at the service level, e.g., by the user. The value of Pointer is internally generated, an object ID or what I've been calling a Unique ID (UID).

There are several reasons why I prefer to think of an association between an Object and an Object ID as being created by an Object generator (not in CLIDT WD 4 but a candidate for WD 5) applied to an instance of a base type. First, one can have Pointers without objects; Allocate is not a characterizing operation of Pointer as Ed has pointed out. Second, the same object identifier can be associated with two different objects in two different contexts. For example, SSN may uniquely identify an object in Table1 and also uniquely identify an object in Table2. (Of course, SSN refers to the same real-life person in both cases.) Pointer1 may point to ObjectA now, to nothing tomorrow, and to ObjectB (of the same type as ObjectA!) the day after tomorrow.

Object (as I use the term in N232) is distinct from Private. First, it has definite characterizing operations, e.g., Assign, Equals. These operations apply to any generated type Object of type T. Second, Object of type T associates a Unique ID (UID) with each instance of the generated type. Pointer to type T can point to such instances. A Dereference of Pointer to type T can obtain the value of type T, the value in the given object. Once one has obtained this value, one can apply to it all the characterizing operations of type T. By contrast, only the originator of a handle can meaningfully perform a Dereference of it.

My reason for regarding Object (and hence generated objects) as entities worthy of consideration at the CLI level is that one can exchange pointers to type T pointing to distinct objects having the same value of type T (but of course different UID's). Examples can range from Pointer to V as an operand of a procedure call to a graphical structure (in the graph-theoretic sense) consisting of objects connected by pointers. In the latter case, sender and receiver should be able to traverse the structure with identical results even though the sending and receiving systems assign UID's to objects independently.

Ed G.

From: Ed Greengrass
To: CLID-ers
Subject: Meek's Two-Quids Worth and Barkmeyers Response - the Sequel

Bit vs. Boolean

I haven't read Meek's paper on the subject but I would distinguish Bit from Boolean as he does. My argument is that their spaces are NOT isomorphic, defining isomorphism as I did in the previous installment. The reason is that while Bit and Boolean are both two-valued, knowing that a given variable has a bit-value of "one" doesn't tell you whether that value corresponds to TRUE or FALSE. On the other hand, knowing that two variables have the boolean values TRUE doesn't tell you whether their product is "one" (TRUE = 1) or "zero" (TRUE = 0). Of course, it is also plain from CLIDT WD 4 that Bit and Boolean have different characterizing operations corresponding to these distinct value spaces.

Table vs. Bag of Record (key, element)

First a minor error. If Table is anything, it is Set of Record (key, element) because CLIDT WD 4 says, reasonably, that "no value of the key datatype can appear in more than one pair."

It seems to me that these two value spaces ARE isomorphic. Moreover, tables (more precisely, relations) are commonly said to have a set-theoretic foundation, which emphasizes the natural connection between the two types. Given that the value spaces are isomorphic, it is plain that any characterizing operation that can be defined for the first type and implemented for any valid representation of the type can also be defined and implemented for the second type. Hence, it seems clear that Table CAN be viewed as Set of Record (key, element).

However, there is another hurdle. I agree with Meek that it is not enough to demonstrate that the characterizing operations of one type can be derived from the characterizing operations of another and vice versa. There is still the question of whether they SHOULD. If the derivation is strained, then it is more natural to think of the two types separately and therefore they ought to be viewed that way too. (Actually, Meek speaks of "adding constraints or removing or adding operators; this seems to imply that the value spaces in question are NOT necessarily isomorphic in which case there should be no question of keeping the types distinct.) Is it unnatural to relate the operations of Table to the the operations of Set of Record (key, element)?

My feeling is that it is natural to distinguish certain aspects of the two groups of characterizing operations but that nevertheless they are related in a fairly natural way. Therefore, it may be quite reasonable to specify Table as Set of Record, etc, i.e., as a derived type, in CLIDT. However, in some cases, one may not want to derive the operations of Table from the operations of Set of Record. Example: A fundamental Table operation selects a table "row" by key value. To define this operation in terms of Set of Record is not hard but it requires something like allowing a value of a record to be specified as "name-of key-field (key-value), *" where "*" has the conventional UNIX meaning of "any value at all" applied to all the remaining fields of the record. Then the Set operation Select becomes Select (name-of-key-field (key-value), *) which is the Table select operation. This is not hard but do we really want to do it that way? Or do we prefer to define the familiar Table Select operation directly?

From: Ed Greengrass
To: CLID-ers
Subject: Meek's Two-Quids Worth and Barkmeyer's Response - the Climax (?)

Barkmeyer has raised the following question: "[W]hen two types have isomorphic value spaces, and one has a subset of the operations of the other, are they distinct types in some primitive way, or is it sufficient to describe the second as derived from the first by eliminating operations?"

The general answer seems to be: T2, a subtype of CLI type T1, should be treated as a distinct, non-derived CLI type if subtyping introduces a significant new conceptual perspective. Integer, a subtype of Real, illustrates the point perfectly. We ask questions of Integer that we do not ask of Real, e.g., the predicate Prime?. Further subtyping to Non-Negative-Integer, introduces a major branch of mathematics, number theory. Hence, not only do we want Integer to be a distinct subtype; the effect of subtyping is to ADD meaningful operations, though perhaps they are not characterizing operations and COULD be defined on Real too.

On the other hand, we may want Scaled to be a distinct CLI datatype because Currency, an extremely important and widely used application type, is most naturally derived from Scaled. Hence, one conclusion might be THAT THERE IS MORE THAN ONE REASONABLE CRITERION FOR INCLUDING A DATATYPE IN CLIDT.

Finally, Ed says, "Many have argued that the computational datatype is "floating-point" or the "scientific number". I am coming to the belief that the mathematical Real datatype has two computational analogues: floating-point and Scaled. And I think perhaps that that is what the CLID should say." I agree except that I would add Rational as a third "computational" analogue. I know that hardly any systems support Rational but it seems to me that Floating Point (AKA "scientific" notation), Scaled, and Rational represent the three principal ways that users "think" about Real numbers (other than integers). In other words, if user or program P sends numeric data to procedure or service S for mathematical or computational services, he (it?) is likely to think of his data in one of those three ways. Whether the underlying hardware or math library software uses floating point is beside the point from a CLIDT perspective. What is not beside the point, however, is that the programmer/numerical analyst who writes S needs to know certain things about the numeric data (e.g., relative error?) in order to calculate properly. This would remain true regardless of what scheme the underlying hardware used to do its arithmetic.

In this regard, consider the way CLIDT WD 4 treats Scaled. Note 3 says that "Scaled is neither a subtype of datatype Real or a proper subtype of datatype Rational. ... [A] scaled datatype is not closed under rational multiplication" This is true only because Scaled is a family of datatypes with each member of the family defined by a radix and scale factor. Rational multiplication of two Scaled values without rounding will produce a new Scaled value which may belong to a different family (different scale factor). This is a computational consideration. It is also the way we users THINK about scaled values. The interest paid on my money market account mothly statement ALWAYS shows two decimal places regardless of the current interest rate.

From: Ed Greengrass
To: CLID-ers
Subject: Meek's Two-Quid's Worth, Etc. - Oh No, Not Again!

Like the monster in the horror movies who comes back again in a new (and more dreadful) sequel however totally demolished he was in the last one (and however much the patient fans were assured that this was really the end), I return with further comments on the Meek-Barkmeyer Two Quid's Worth discussion.

Bit String vs. Bit Map vs. List of Bits

Once again, Brian has raised a several very good points: (1) Bit-strings are not very list-like, e.g., insertion and deletion of bits are uncommon, (2) Bit-strings have logical operations such as "mask" and "shift" that are not list-like at all (though of course they can be defined on lists), and (3) Bit-strings (as Brian says, more properly called bit MAPS in this context) can be used to represent multi-dimensional objects, e.g., two-dimensional images, three-dimensional holographs, etc. Let me see if I can clarify these astute observations.

In essence, bit-strings fall into two categories ACCORDING TO HOW THE BITS ARE ACCESSED:

Category 1: The bits correspond to an ordered collection of "real-world" objects, e.g., they may correspond to disk segments with "one" meaning allocated and "zero" meaning unallocated. In that example, bit 1 corresponds to segment 1, bit 2 corresponds to segment 2, etc. Hence the ordering of bits corresponds to the ordering of the objects represented and a given object (or bit) is accessed by ordinal position. Hence, this kind of bit-string may be defined as "ARRAY OF BIT". It is properly called a bit-MAP (as Brian says) because there is an implicit mappping beteen the bits and the objects to which they correspond. It can be multi-dimensional (also as Brian says). In the case of an N-dimensional image, the bits represent a multi-dimensional array and are ordered by physical position in N-space, i.e., each bit corresponds to a small region of space and determines the presence or absence of light in that region.

Category 2: The bits correspomd to attributes of a "real-world" object or set of related objects. For example, bit one means "I/O interrupt status", bit two means "CPU exception status", etc. In this case, bit ordering is not important. Bits are identified by NAME where the name is a mnemonic for the meaning. In the bad old assembly language days, we sometimes made an explicit reference to, e.g., bit four, but the "4" was really just an encoding for what the bit "really" meant, e.g., "device ready status". (Naturally, no well-brought up young programmer would hard-code a "4" today!) Hence, this kind of bit-string may be viewed as a RECORD where each field is of type BIT and the field names correspond to the meanings of the individual bits.

(One might be tempted to define category 2 bit-strings as Tables. Each row of the table would consist of a fixed length Character String key and a Bit type element. The key would be named something like "Meaning Of Bit" and key values would correspond to the names of the bits. The trouble with this kind of definition is that the bit-strings tend to be fixed length whereas tables are variable length and can even be empty.)

One important point about BOTH bit-string categories should be noted. The individual bits might easily be bit sub-strings or something more complex , e.g., in category 1, each "pixel" of an image might be an Integer corresponding to gray level or color. The essential point is that at the CLI level, bit-strings are essentially MAPS. Commonly, they are bit maps because we frequently want to track binary attributes, e.g., allocated vs. unallocated, set vs. reset, full vs. empty, on vs. off, etc. However, they remain MAPS even when the attributes being tracked are not binary.

Conclusion

Looking back at this message and its three "two quid's worth" predecessors, I find that I agree with much that Brian says but with one significant difference: Brian would make types like Bit-Map and Character String primitives. I have treated them as derived types because I found the derivations illuminating. In particular, it led me to view Character String as two distinct types, and ditto for Bit-String. (Brian spoke of Character String as a hybrid.) However, there is this to be said for treating them as Primitives: it gives them the prominence and fundamental importance that they probably deserve. In a previous message, I suggested a compromise: define them as derived types but define their characterizing operations directly if defining them in terms of the characterizing operations of their base types would lead to unnnecessary complexity.