Date: Fri, 15 Mar 91 17:50:17 EST
From: edbark@cme.nist.gov (Ed Barkmeyer)
Subject: Re: Craig's Comments on many comments

I agree with the core-type/extended-type idea, and I believe that most of us do. The issue is, as always: where do you draw the line? BitString is "enough to be representationally complete". After Character and Integer, which seem to be universally acceptable, literally every other primitive datatype has come under fire, as either not absolutely necessary in the core or not defined in the core to be the type wanted. Similarly, among generators, everyone agrees on some sequence notion, but that's where the agreement ends; and almost everyone (except SQL) agrees on Record.

The problem with the standard "external representation" is that implementors will inevitably see it as the "required exchange form" for the datatype. For RPC, the true "required exchange form" will in fact be defined by the RPC protocol in the ASN, and not by the CLID AT ALL. For CLIPC locally, the "required exchange form" is platform-dependent, and is subject to vendor- defined standard, where the hardware vendor is in a position to dictate it, and to "additional standardization", where no one vendor is in such a position, as in the IBM PC. The CLID and the CLIPC must unequivocally say this!

In my view, there will be four levels of type definition: the CLID "core", the CLID "extended datatypes", the "extended datatypes" defined by other standards, and the "extended datatypes" defined by users and individual applications. The advantage of the extended types defined by the CLID itself, and those defined by other standards well-known to the RPC/CLIPC implementors (such as POSIX), is that these may have "required exchange forms" which are NOT derived from their "external representation" (their CLID type-definition), while the extended types defined by other (application) standards and users must necessarily have an exchange form derived from the "external represent- ation". It is logically from the third pool, the datatypes defined by other standards, that the additions over time to the CLID extended datatypes will be drawn. But there will also always be types in the third pool.

In re-examining the datatypes with this view in mind, I would probably put Rational and Complex in the "extended" category, since the ordered-pair of Integer/Real, is a computationally adequate and apparently standard and optimal representation.

I have stronger reservations about doing this to Scaled, in which Craig's proposed definition requires considerable semantic additions and is unrelated to the exchange form. Craig's technique for Scaled could be applied to Real just as well, and the reasons why we don't want to do this to Real are the same ones for not doing it to Scaled. They are primitive mathematical datatypes which have independent computational notions and hardware support.

And I flatly reject the definition of Date-and-Time as a CharacterString. No aspect of the semantics of Date-and-Time, nor any of its characterizing operations, are remotely related to those of CharacterString. This definition is purely representational, and is equivalent to: TYPE Real = BitString! There is room, however, for defining Date-and-Time as a Scaled, or Real or Integer, type, with semantics of offset in seconds (or whatever) from a fixed point in history, a la Paul's document and many hardware clocks.

I almost agree with Craig about Set and Bag. The minimum (core) generators are: Record, Choice and Bag. List is a Bag with ordering; Set is a Bag with uniqueness; Array is a List with subscript-mapping; and Table is a Bag with some other kind of associative map. Record and Choice, on the other hand, are entirely different things. Pointer might be better defined as the State type which it really is, although I doubt it, and its relationship to the Table concept or the Heap (= Bag of Any) concept could be explored.

The same kind of argument applies to State and Enumerated. Enumerated is a State with ordering.

And Boolean could be a two-valued State-type with NOT and AND (add AND to Switch). I've never been sure what Bit is.

Re:

(11) By "scientific number", I mean what every kid is taught it means in Physics, Chemistry and other natural sciences - a value which is only correct to so many significant digits, which is what you get from slide-rules, floating-arithmetic and many measurement systems. Real(rel-err) is exactly this notion, in which rel-err specifies the number of significant digits. This is as opposed to "absolute error" or "tolerance", which occurs with at least equal frequency in many engineering disciplines, and is properly represented by Scaled, at least to the same degree that "significant digits" is equivalent to "relative error".

(13) Null is not Undefined, and they are both States in some sense. You may be able to avoid Null or Undefined values in procedure calling by complicating the concepts of "result" and "return", but the same problem appears in elements of Records and Arrays, where such subterfuges are not available. I have argued that you need to keep Null (= State(null)), because it is not an error in itself, whereas Undefined probably is.

(14) The distinction Craig makes between Private/Opaque and Octet/BitString is correct. This indicates that the text of the CLID under Private which indicates that it does double-duty may be an indication that there should be two datatypes there:
a) Private/Opaque has a value-space which is intentionally concealed from all processing entities except the "package" which owns it, implying that it is meaningless to the calling program as well; while
b) Octet/BitString has a value-space whose bit representation must be preserved through intermediaries but is meaningful to both the calling program and the called program, and "not expressible as a CLI datatype."
Dan Yellin's observation that Private and Octet both refer to an "uninterpreted string of bits" is therefore not quite right. The string of bits IS interpreted, either by the called program only or by both parties. Does the distinction between "one of" and "both" require a distinction in datatypes? The bits are in either case "visible to intermediate recipients", but presumed to be meaningless to them in both cases.

(16) Other subtype forms. I think the syntax for Selected ought to include Ranges, although conceptually disjoint ranges are just shorthands for selection-lists. Single Ranges are useful in applications; Selected is just yet-another-constraint. Extended is apparently there for the guys who want to put Infinity values in the Real datatype, which apparently includes the RPC community, and other such gems as Infinitesimal (why not?). More importantly, it supports "upward compatible" State and Enumerated datatypes, which is the "core" vs. "extended" philosophy applied to values, rather than types.

Subtyping Lists (et al.) to bound the number of values is important to applications, and formalizes, thereby encouraging documentation of, a common implementation requirement. (I disagree with Ed Greengrass that an implementation is free to bound a datatype which a standard indicates is unbounded. If it is the intention of the standard that the datatype can be bounded in implementations, then the standard should so specify. Both Pascal and Fortran clearly contain the notion that the Integer datatype is bounded in all implementations, but Pascal enunciates and parametrizes the notion, while Fortran buries it in the discussion of size of values for COMMON and EQUIVALENCE.) Many languages permit conforming implementations to limit the length of character strings, for example, without providing a requirement or means for the documentation of that limit.

The general problem of subtyping any generated datatype by subtyping its components, Records in particular, has significant impact on conformance of non-primitive procedure parameters. This is one of those areas in which programming languages have not yet caught up with an age-old concept. C-programmers use tricks (which lint always complains about) to accomplish what the language does not support, while Ada and Pascal programmers always use only the base type with variants, even when almost all usages are of one subtype, and other languages often force the programmer to use two different datatypes and put the subtype semantics in the code.

The problem of subtyping Procedure types (or procedure template types) is messy and needs work. But because procedures can be parameters to procedures, and in some languages referenced by Pointer

datatypes, the problem of subtype conformance applies here also.

(*) I resorted to productions for "constants" in CLID precisely in order to support the representation of subtyping parameters. It is clearly necessary to have a means of expressing values of an arbitrary primitive datatype, in order to support Range or Selected. As a free by-product, you get a means of expressing integer values to parametrize Scaled and Time, and to limit the sizes of List/Set/Bag/Table. (Since the CLID currently requires the subscript-range of arrays to be a Datatype, it is covered under Range.) One also needs to express values of any generated datatype, in order to support Selected (and possibly Range for defined-types which have ordering). This is a "completeness" issue, not a "usefulness" issue.

(25) Syntactic distinction between type-generators and parametrized datatypes is unnecessary, if I can get around it in the text. But the semantic distinction is quite significant. Parametrization is a means of identifying distinct primitive types, for which the primitive-ness is an important property, and a means of defining subtypes, for which the subtype-ness is an important property. Generators create datatypes which are NOT primitive and NOT subtypes. The problem that motivated the syntactic distinction is the attempt to capture semantic rules in the syntactic elements, so that you don't get "dead-end" productions, or additional rules like:

"The actual parameter corresponding to a formal parameter in a type- definition must be a datatype or a value according to whether the formal parameter appears in contexts requiring a datatype or a value." and

"A formal parameter in a type-definition shall appear either in contexts requiring a datatype or in contexts requiring a value, but not both."

The proposed IDN syntax rules allow, for example:

```
TYPE boojum(snark) = Record(
                        snark: Integer,
                        first: snark,
                        control: State(isboo, notboo, snark)
                      )
```

This problem did not occur with "parametric-datatype" and "parametric-value". I need to look at this.

On Section 4. I wholeheartedly agree. This means the IDN must allow one to IMPORT definitions and add annotations to the imported objects. Since the current syntax, however, distributes annotations over several elements of a declaration, this is no mean task.

-Ed