From: edbark@cme.nist.gov (Ed Barkmeyer)
Subject: Suggested CLID and IDN changes per Arles

I had written the following before Arles, but I forgot to bring the copy with me. After our discussion, I mentioned to Dan Yellin that I already had text for some of the things we agreed to. I FAXed this to the RPC group on Thursday morning (30 May). I thought you would all like to have seen it.

-Ed Barkmeyer

Recommend Changes to the IDN and CLID WD5 per Arles Discussions

I. Suggested syntax changes and corresponding text changes for areas agreed to in Arles.

1. Some subtype constructions are ambiguous. E.g. Octet:

Octet = list of bit: size(8) is ambiguous. Is it list of (bit:size(8)) or (list of bit):size(8) ? While size(8) can only reasonably modify "list", the syntax does not make that obvious, and a completely ambiguous example is:

list of set of character(ISO8859_1): size (8) Is it the list or the set which is being restricted to size 8?

Suggested change:

```
pointer-type = "pointer" "to" "(" base ")" .
list-type = "list" "of" "(" element ")" .
set-type = "set" "of" "(" element ")" .
bag-type = "bag" "of" "(" element ")" .
array-type = "array" "(" array-index-list ")" "of" "(" element ")" .
table-type = "table" "(" key ")" "of" "(" element ")" .
explicit-subtype = "restricted" "to" "(" subtype-definition ")" .
```

This change delimits the "element" datatype with parentheses, so that subtypes which belong to the element-type must appear within the parens, while those belonging to the generated datatype appear outside them.

2. Array and table-types need an additional change, to change the brackets [] to parens (). In addition, the mulidimensional array should have all its indices in a single list.

Suggested change:

```
array-type = "array" "(" array-index-list ")" "of" "(" element ")" .
array-index-list = array-index { "," array-index } .
array-index = index-type | lowerbound ".." upperbound .

table-type = "table" "(" key ")" "of" "(" element ")" .
```

3. Late-bindings. Rather than the use of "*" to denote dynamically defined array-bounds, only for arrays and only in procedure arguments, there should be a more general means of supporting datatypes which are selected from a common "family" by a "dynamically defined" parameter. In addition to "conformant arrays", there is the COBOL "OCCURS ... DEPENDING ON x" construct, in which the size of a "list" is specified by another field of the Record-type which contains the list, and the Pascal "tag-variable : tag-type" construction, which specifies how a "variant record" (choice-type) is discriminated, and several other such constructs in various programming languages.

The proposed IDN syntax allows for, indeed requires, the specification of the source of the dynamically defined values in an "attribute" construction. Since the draft says that "attributes" do not add semantic information to the datatype definitions, this is a dubious use of attributes. When the needed value is the value of another component of the a common datatype, that appears to be useful semantic information in

the description of the common datatype.

Suggested change:

To the production for value-expression, add the alternative "late-binding". Then add the following clause, assuming that the term "generated datatype" includes both Pointer and Procedure (see recommendation 4 below):

x.x.x Late-Bindings

A late-binding identifies a value which is the value of another component of some generated-datatype in which the late-binding occurs. Syntax:

late-binding = late-binding-primary { "." component-reference } .
late-binding-primary = field-identifier | argument-name .
component-reference = field-identifier | "*" .

A datatype-designator x is said to "involve" a particular instance of a late-binding if x contains the instance and there is no component datatype y of x which contains the instance. Thus, exactly one datatype-designator "involves" a given instance of a late-binding. Any datatype-designator which involves a late-binding shall identify a component of some generated-datatype.

The late-binding-primary shall identify a (different) component of some generated-datatype which contains the datatype which involves the late-binding. The component so identified is said to be the "primary component", and the generated datatype of which it is a component is said to be the "primary datatype". The primary datatype shall be either a procedure-type or a record-type. When the primary datatype is a procedure-datatype, the late-binding-primary shall be an argument-name and shall identify an argument of the primary datatype. When the primary datatype is a record-type, the late-binding-primary shall be a field-identifier and shall identify a field of the primary datatype.

When the late-binding contains no component-references, the value of the late-binding shall be the value of the primary component. Otherwise, the "current reference" shall comprise the late-binding-primary, and identify the primary component and its value, and the following paragraph shall be applied (recursively) to determine the value of the late-binding.

The datatype of the current reference shall be a record-type, a choice-type, or a pointer-type. If the datatype of the current reference is a record-type or a choice-type, then the next component-reference shall be a field-identifier and shall identify a field of the current reference. In this case, a new reference shall comprise the current reference syntax, plus the next component-reference, and its value shall be the value of that field of the current reference which is identified by the next component-reference. If the current reference is a pointer-type, the next component-reference shall be an asterisk. In this case, a new reference shall comprise the current reference syntax, plus the asterisk, and its value shall be that obtained by dereferencing the pointer-value of the current reference. In either case, when the late-binding contains no further component-references, the value of the late-binding shall be the value of the new reference. Otherwise, the new reference shall become the current reference and the rules of this paragraph shall be applied thereto.

NOTES:

1. The datatype which involves a late-binding must be a component of some generated-datatype, but that generated-datatype may itself be a component of another generated-datatype, and so on. The primary datatype may be several levels up this hierarchy.

2. The primary component, and thus the primary datatype, cannot be ambiguous, even when the late-binding-primary identifier appears more than once in such a hierarchy. By the scope rules, the meaning of the identifier X in a datatype-designator (y) which involves X as a late-binding-primary is the meaning of X as declared in y, or else the meaning of X in the datatype-designator which immediately contains y. So, in following the hierarchy upward, there is a "most recent declaration" of X, and that is the one to which the late-binding refers.

3. In the same wise, an identifier which may be either a value- identifier or a late-binding can be

resolved by application of the same scope rules. If the identifier X is found to have a "declaration" anywhere within the outermost datatype-designator which contains y, then that declaration is used. If no such declaration is found, then a declaration of X in a "global" context, e.g. as a value-identifier, applies.

4. Generated-datatypes again. It appears that the previous WG11 effort to distinguish generated-datatypes from primitive-datatypes, and the current text of the draft in that regard, was made without careful attention to the Pointer and Procedure datatypes. (Indeed, it was decided while the nature of Pointer was being debated and Procedure had not yet been added.) This has resulted in some recent questions asking whether Pointer and Procedure are primitive, generated, or something different altogether.

It seems to me that the last is correct. More accurately, all of the current "generated-datatypes" save Pointer are really "aggregate" datatypes. The values of List, Set, Bag, Array, Table, Choice, and Record datatypes are physical, or at least logical, composites of values of their "element" datatypes. This is not so of Pointer. The text even says that the values of Pointer-types belong to some primitive State-type. And it is clear that values of Procedure-types are, as the text says, primitive.

The problem is that Pointer and Procedure do have something in common with the aggregate datatypes, namely "generation from" (or "parametrization by") "component" (or "argument") datatypes. And it is also clear that the only way a user can define a "procedure-type family" is by use of a "generator-declaration". This has the interesting anomaly that datatypes defined expressly by procedure-type syntax are "primitive", whereas datatypes defined by use of a generator whose generation-template is a procedure-type are "generated". It seems that, as Craig Schaffert has said, the distinction between primitive and generated is syntactic!

I would argue that "generated" datatypes SHOULD BE a syntactic classification, namely those datatypes which have "component" or "parametric" datatypes, for whatever reason. If there is a need to distinguish values which are "primitive" from values which are "aggregate" or "composite", then such a terminology should be used, and that classification should be distinct from "simple" versus "generated". All aggregate types are necessarily "generated", but not all generated types are aggregates. And conversely, all simple types are necessarily primitive, but not all primitive datatypes are simple.

Suggested changes:

Since generated-datatype is not a semantically interesting distinction, the production for CLI-datatype in Clause 7 should read:

CLI-datatype = primitive-datatype | aggregate-datatype | subtype
| defined-datatype | defined-generator-type .

Pointer and Procedure should both be in 7.1 (Primitive datatypes), and 7.3 should be renamed "Aggregate datatypes" (or "Composite datatypes"). 7.3.?? (defined-generator-type) should then become 7.5.

I would further suggest that the term "generated-datatype" is only useful with respect to type-definition and the syntactic structure of datatype-designators, and should therefore be introduced in 7.4 or 7.5 or 8. (It may be that "generated-datatype" can replace "defined- generator-type"). Alternatively, it may be better introduced in 6.?? along with "family".

5. Noise words. The purpose of the IDN syntax is to provide for purely formal declarations of interfaces. Noise words add nothing but processing time. The noise words "of" and "to" should be removed, or at least made optional, especially after change 1 above.

Suggested change:

pointer-type = "pointer" ["to"] "(" base ")" .
list-type = "list" ["of"] "(" element ")" .
set-type = "set" ["of"] "(" element ")" .
bag-type = "bag" ["of"] "(" element ")" .
explicit-subtype = "restricted" ["to"] "(" subtype-definition ")" .

Note: There does seem to be legibility added to array and table definitions in which the keyword "of" separates" the two groups of parenthesized datatypes (after change 2 above).

II. Recommended Semantic changes

6. Choice-types should support a "discriminator" concept, so that the C/ASN.1 style of (barely) discriminated union is not the only one which can be supported. The Pascal/Ada style needs to be supported as well. [U.S. position]

Suggested change (syntactic variance from U.S. proposed syntax):

```
choice-type = "choice" "(" selection-type [ "=" discriminant ] ")"
                "of" "(" alternative-list ")"  .
selection-type = datatype  .
discriminant = value-expression  .
alternative-list = alternative { "," alternative }  .
alternative = selection-value ":" alternative-type  .
selection-value = select-list | "*"  .
alternative-type = datatype  .
```

The select-list is, as in "Selecting" subtypes, a list of (ranges of) values, and the "*" means "any value of the selection-type which does not appear in an alternative". The use of the select-list implies that the restriction to discrete datatypes (of Selecting) should also apply to the selection-type. This form permits the Pascal/Ada variant to be representable, and allows the C-type discriminated union to be supported by a slight subterfuge. E.g.:

```
choice( state(a1, a2, a3) ) of (
   a1: real,
   a2: integer,
   a3: boolean
)
```

The useful forms of value-expression for the discriminant are late-binding or parametric-value which resolves to late-binding, but there is no reason to exclude the use of constants. Unlike Pascal and Ada, the discriminant FOLLOWS the selection-type to be consistent with the unwritten CLID rule that the datatype of a value-expression must be known before the value-expression is encountered. Strictly speaking, this is only necessary because constants are allowed. An alternative syntax is:

```
choice-type = "choice" "(" [ discriminant ":" ] selection-type ")"
                "of" "(" alternative-list ")"  .
discriminant = late-binding | parametric-value .
```

The only problem with this approach is that unlike fields of a record, fields of a choice will not be referenceable. This loses information which was required in all of C, Pascal and Ada.