How to Handle Missing Information Using S-by-C

by Hugh Darwen and Erwin Smout

Review draft, 05 August 2008

1. Introduction

In an e-mail entitled **How to handle missing information without using nulls,** to the forum that discusses reference [1] (ttm@thethirdmanifesto.com) on 20 February 2008 at 17:53, Erwin Smout wrote:

> The following idea crossed my mind yesterday:
>
> For each and every type xxx, the system automatically provides a proper supertype, called e.g. POSSIBLYMISSINGxxx. There is precisely one value that is of the supertype but not of the type itself, and that value is the value 'MISSING' (yes, do let me consider it a value for the time being).
>
> All the operators defined for the type do apply only to the type xxx itself, and thus not to type POSSIBLYMISSINGxxx. The only operator that does apply to type POSSIBLYMISSINGxxx is, optionally, an operator ISMISSING() or ISVALUE(), returning a boolean according to the obvious semantics (having this operator is not strictly necessary because an equality test to the value MISSING does the job too). Type checking thus prevents the invocation of, say, the + operator on expressions of type POSSIBLYMISSINGxxx.
>
> Relation attributes can be declared to be of type POSSIBLYMISSINGxxx, or of type xxx. In the case an attribute is declared to be of type POSSIBLYMISSINGxxx, the programmer is forced to first invoke the ISVALUE() test before invoking a TREAT_AS_xxx (if he wants to build stable, reliable programs and avoid runtime errors, that is).
>
> There would be one 'MISSING' for each "root" type (which, by the creation of the proper supertype, no longer is a "genuine" root type, but that isn't much of a problem, I think). Let's say that type INTEGER = {-32768 ... 32767}. A proper subtype of INTEGER is EVENINTEGER = {-32768, -32766, ... 32766}. POSSIBLYMISSINGINTEGER = {-32768 ... 32767} U {MISSINGINTEGER}, and POSSIBLYMISSINGEVENINTEGER = {-32768, -32766, ... 32766} U {MISSINGINTEGER}.
>
> Such that if types xxx and yyy do not have a common supertype (alpha notwithstanding), then neither do the types POSSIBLYMISSINGxxx and POSSIBLYMISSINGyyy. Likewise, if yyy is a proper sub/supertype of xxx, then POSSIBLYMISSINGyyy is a proper sub/supertype of POSSIBLYMISSINGxxx.
>
> The value MISSING of POSSIBLYMISSINGINTEGER is equal to the value MISSING of POSSIBLYMISSINGEVENINTEGER, with the obvious consequences for join, duplicate elimination in projections, etc. etc.
>
> It looks to me like this is a relationally sound solution for the 'missing information' problem. Does it look like a sensible idea to try and work this out in code?

The answer to Smout's final question struck Darwen as being "yes", at least if the method of subtyping is one based on specialization by constraint (S-by-C), as proposed in [1]. Accordingly, he set to work on the geometrical examples (ellipses and circles, rectangles and rhombuses) that are used in [1], to see how `POSSIBLYMISSING` counterparts of those type

schemas might be done in **Tutorial D**. The purpose of this paper is to present the results of these case studies and comment on them.

2. The Case Studies

In what follows:

1. The abbreviations *TTM* and *IM* refer to Chapters 4 and 13 of reference [1], respectively. (Certain "IM Prescriptions" mentioned in our text are given in *IM*, of course.)
2. Smout's `POSSIBLYMISSING` is abbreviated to `PM_`. Thus, the "possibly missing" counterpart of type `INTEGER` would be `PM_INTEGER`, of *`alpha`* `PM_`*`alpha`*, and so on.

2.1 Ellipses and Circles

Reference [1] uses the example of ellipses and circles to illustrate subtyping with *single inheritance* only. Single inheritance applies when no type under consideration has more than one *immediate supertype*. Some ellipses, but not all ellipses, are circles, and a circle has a radius whereas an ellipse in general does not. Thus it is appropriate to define `CIRCLE` as a *proper subtype* of type `ELLIPSE`. Additionally, when the discussion needs to probe a little further, the type `O_CIRCLE`, for circles centred at the origin, is defined as an immediate subtype of `CIRCLE` (and hence a proper subtype of both `CIRCLE` and `ELLIPSE`).

Here are **Tutorial D** definitions of these three types, assuming `ELLIPSE` to be a *root type*[1] (one with no proper supertypes):

```
TYPE ELLIPSE POSSREP { A LENGTH, B LENGTH, CTR POINT } ;
TYPE CIRCLE   IS { ELLIPSE
                   CONSTRAINT THE_A(ELLIPSE)= THE_B(ELLIPSE)
                   POSSREP {  R   = THE_A(ELLIPSE),
                              CTR = THE_CTR(ELLIPSE) } } ;
TYPE O_CIRCLE IS { CIRCLE
                   CONSTRAINT THE_CTR(CIRCLE)= POINT(0.0, 0.0)
                   POSSREP {  R = THE_R(CIRCLE) } } ;
```

Example 1: Ellipses and Circles as in [1]

Under Smout's proposal we will add to the above schema `PM_` counterparts of each of those three types: `PM_ELLIPSE` for ellipses plus the value `'MISSING'`[2], `PM_CIRCLE` for circles plus `'MISSING'`, and `PM_O_CIRCLE` for O-circles plus `'MISSING'`.

Notice first of all that `PM_ELLIPSE` must be a proper supertype of `ELLIPSE`—it contains all the values of type `ELLIPSE` plus one value, `'MISSING'`, that is not a value of type `ELLIPSE`. And the fact that `PM_ELLIPSE` has just one value that is not of type `ELLIPSE` clearly rules out the possible existence of a type that is a proper supertype of `ELLIPSE` and a proper subtype of

[1] In [1] `ELLIPSE` is a subtype of root type `PLANE_FIGURE`.
[2] The term Smout used in his e-mail, but in fact we shall not need this term, as will be seen.

PM_ELLIPSE. So ELLIPSE has to be an immediate subtype of PM_ELLIPSE. So we might think that our revised schema has to start like this:

```
TYPE PM_ELLIPSE POSSREP { ??? } ;
TYPE ELLIPSE IS {  PM_ELLIPSE
                   CONSTRAINT ISVALUE(PM_ELLIPSE) } ;
```

But this approach cannot work! If PM_ELLIPSE has possrep components, then it must support operators implied by the definitions of those components, under RM Prescription 5 of *TTM*. Fortunately, **Tutorial D** has a solution to this problem, in accordance with IM Prescription 20 of *IM*, concerning *union types.* A union type does not require to be defined with a possrep because, by definition, every one of its values is a value of at least one of its immediate subtypes. Accordingly, we can define a type *M*, containing just the value 'MISSING', and make PM_ELLIPSE the union type for *M* and ELLIPSE:

```
TYPE PM_ELLIPSE UNION ;
TYPE ELLIPSE IS {  PM_ELLIPSE
                   POSSREP { A LENGTH, B LENGTH, CTR POINT } } ;
TYPE M IS { PM_ELLIPSE POSSREP { } } ;
```

Example 2: Introducing types PM_ELLIPSE and *M*

Points to note:

1. Under this approach a separate "MISSING" type *M* must be defined in connection with every root type that is to include 'MISSING', as mentioned by Smout. Perhaps the definition of *M*, including its actual type name, can be implied by the type name prefix PM_—or, more likely, by some special key word given in the type definition of a PM_ type. Note, however, that if we accept *alpha* as the only root scalar type, then just one *M* suffices. Moreover, that one *M* could be a built-in type, in which case *alpha* need exist only conceptually.

2. The possrep for *M* is empty, this being the easiest way of defining a singleton type. Thus, there are no THE_ operators for *M* and its single *selector*,[3] also named *M*, has no parameters. The value that Smout refers to as 'MISSING' is then denoted by the selector invocation *M*( ), and his ISMISSING operator becomes the IS_*M* required by IM Prescription 15 of *IM*.

3. PM_ELLIPSE is not only a union type but also a *dummy type*, having no possrep. Thus, a literal of type PM_ELLIPSE can be expressed only by invoking a selector for one of its regular subtypes. *M* and ELLIPSE, having possreps, are both regular.

Now we must figure out how to incorporate the types PM_CIRCLE and CIRCLE into the schema.

Clearly CIRCLE has to be an immediate subtype of PM_CIRCLE, but what about PM_CIRCLE itself? Each of its values is either *M*( ) or a value of type ELLIPSE that has equal semiaxes. Obviously it is not a subtype of *M*, but nor is it a subtype of ELLIPSE, because *M*( ) is not a

[3] see RM Prescription 4 in *TTM*

value of type ELLIPSE. We have to conclude that it is an additional immediate subtype of PM_ELLIPSE. But if it is an immediate subtype of PM_ELLIPSE, then *M* cannot also be an immediate subtype, because *M( )* is a value of type PM_CIRCLE. These considerations lead us to

```
TYPE PM_ELLIPSE UNION ;
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } } ;
TYPE PM_CIRCLE UNION
               IS { PM_ELLIPSE
                    CONSTRAINT ??? } ;
TYPE M IS { PM_CIRCLE POSSREP { } } ;
```

Example 3: Introducing type PM_CIRCLE

Once again we must avoid defining a possrep for the PM_ type, the reason being as with PM_ELLIPSE—under Smout's proposal no THE_ operators must be defined for a PM_ type. Therefore PM_CIRCLE must be a union type (of *M*, and the type CIRCLE to come later) and must be defined by a constraint on the values of type PM_ELLIPSE. In **Tutorial D**, such a constraint must be defined using operators defined for the supertype, in this case PM_ELLIPSE. We need that constraint to admit *M( )*, every value of type ELLIPSE that has equal A and B components, and no other values. But to test for equal A and B components we need the operators THE_A and THE_B, which are defined for type ELLIPSE but not for type PM_ELLIPSE. Our next try, then, is this:

```
TYPE PM_ELLIPSE UNION ;
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } } ;
TYPE PM_CIRCLE UNION
               IS { PM_ELLIPSE
                    CONSTRAINT IS_M(PM_ELLIPSE) OR
                               WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E :
                               THE_A(E) = THE_B(E) } ;
TYPE M IS { PM_CIRCLE POSSREP { } } ;
```

Example 4: Type constraint for PM_CIRCLE

Points to note:

1. WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E merely saves us having to repeat the invocation TREAT_AS_ELLIPSE(PM_ELLIPSE). The expression E has declared type ELLIPSE.

2. The definition relies on a left-to-right evaluation of expressions of the form *a* OR *b*, such that the system will not attempt to evaluate the condition *b* unless condition *a* is FALSE (TREAT_AS_ELLIPSE gives a run-time error when the operand is not a value of type ELLIPSE). Such a mechanism is commonly found in computer languages but is

frowned upon by some purists. We assume that some workaround[4] is available in a system that does not support this mechanism.

3. The immediate subtypes of PM_ELLIPSE are no longer disjoint. The examples of union types given in [1] are all such that their immediate subtypes are disjoint, but the definition of union type does not require this to be so. For example, type PARALLELOGRAM could be defined as a union type with immediate subtypes RECTANGLE, RHOMBUS, and NONSPECIAL for all the parallelograms that are neither rectangles nor rhombuses. RECTANGLE and RHOMBUS are of course not disjoint.

4. *Problem:* the definitions of *M* and PM_CIRCLE reference each other: *M* is an immediate subtype of PM_CIRCLE and PM_CIRCLE's type constraint invokes IS_*M*, an operator that comes into existence with the definition of type *M* (see *IM*, IM Prescription 15).

A solution to the problem described in point 4 is for the language to support, and the *IM* to require it to support, multiple type definitions in a single statement. A precedent for such an approach has been set by *TTM*'s requirement, and **Tutorial D**'s support, for *multiple assignment*, whereby several variables can be simultaneously updated in a single statement. A multiple assignment is written as a sequence of individual assigns, with commas separating them instead of semicolons. Clearly we can take the same approach with type definitions, replacing the problematical example 4 by

```
TYPE PM_ELLIPSE UNION ,
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } } ,
TYPE PM_CIRCLE IS { PM_ELLIPSE
                    CONSTRAINT IS_M(PM_ELLIPSE) OR
                               WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E :
                               THE_A(E) = THE_B(E) } ,
TYPE M IS { PM_CIRCLE POSSREP { } } ;
```

Example 5: All types defined in one statement

—note the commas.

Notice now that the values of type CIRCLE are precisely those values of type PM_CIRCLE that are also values of type ELLIPSE. Therefore we add CIRCLE to the schema thus:

[4] For example, CASE WHEN ISMISSING(PM_ELLIPSE) THEN TRUE ELSE WITH … etc. END CASE. Although logically equivalent to the simpler expression using OR, most authorities accept the use of CASE, whose definition relies on the ordering of the WHEN clauses, as being more acceptable.

```
TYPE PM_ELLIPSE UNION ,
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } ,
TYPE PM_CIRCLE UNION
               IS { PM_ELLIPSE
                    CONSTRAINT IS_M(PM_ELLIPSE) OR
                               WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E :
                               THE_A(E) = THE_B(E) } ,
TYPE CIRCLE IS {  PM_CIRCLE, ELLIPSE
                  POSSREP { R   = THE_A(ELLIPSE),
                            CTR = THE_CTR(ELLIPSE) } } ,
TYPE M IS { PM_CIRCLE POSSREP { } } ;
```

Points to note:

1. As required under Smout's proposal, the operators `THE_A`, `THE_B`, and `THE_CTR` are defined for type `ELLIPSE` but not for type `PM_ELLIPSE`; and `THE_R` is defined for `CIRCLE` and not for `PM_CIRCLE`.

2. Our type schema is no longer a strict hierarchy, because we have used *multiple inheritance* to derive type `CIRCLE`. It seems, then, that the `PM_` counterpart of the simplest possible case of S-by-C cannot be a hierarchy.

Now it is a simple matter to add the types `PM_O_CIRCLE` and `O_CIRCLE` in similar fashion to `PM_CIRCLE` and `CIRCLE`:

```
TYPE PM_ELLIPSE UNION ,
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } ,
TYPE PM_CIRCLE UNION
               IS { PM_ELLIPSE
                    CONSTRAINT IS_M(PM_ELLIPSE) OR
                               WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E :
                               THE_A(E) = THE_B(E) } ,
TYPE CIRCLE IS {  PM_CIRCLE, ELLIPSE
                  POSSREP { R   = THE_A(ELLIPSE),
                            CTR = THE_CTR(ELLIPSE) } } ,
TYPE PM_O_CIRCLE UNION
                 IS {PM_CIRCLE
                     CONSTRAINT IS_M(PM_CIRCLE) OR
                                THE_R(TREAT_AS_CIRCLE(PM_CIRCLE)) =
                                  POINT(0.0, 0.0) } ,
TYPE O_CIRCLE IS { PM_O_CIRCLE, CIRCLE
                   POSSREP { R = THE_R(CIRCLE) } } ,
TYPE M IS { PM_O_CIRCLE POSSREP { } } ;
```

Figure 1 shows the type graph for this schema.

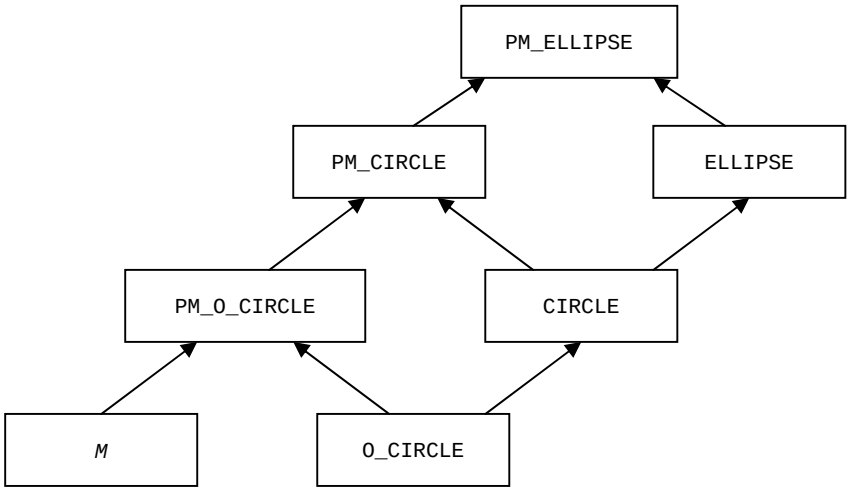

Figure 1: Type graph for PM_ ellipses and circles

2.2 Rectangles and Rhombuses

To illustrate multiple inheritance, reference [1] uses type SQUARE, noting that every square is both a rectangle and a rhombus. Rectangles are special cases of parallelograms, as are rhombuses. The type schema given in Chapter 15 of [1] is like this, with PARALLELOGRAM a root type:

```
TYPE PARALLELOGRAM POSSREP { A POINT, B POINT, C POINT, D POINT
                             CONSTRAINT … } ;
TYPE RECTANGLE IS { PARALLELOGRAM CONSTRAINT … POSSREP … } ;
TYPE RHOMBUS   IS { PARALLELOGRAM CONSTRAINT … POSSREP … } ;
TYPE SQUARE    IS { RECTANGLE, RHOMBUS } ;
```

The constraints and possreps given in [1] for RECTANGLE and RHOMBUS, and the constraint for PARALLELOGRAM, are omitted for convenience, being unimportant for present purposes.

To develop the type definition for PM_PARALLELOGRAM we follow the method used for PM_ELLIPSE—

```
TYPE PM_PARALLELOGRAM UNION ,
TYPE PARALLELOGRAM IS { PM_PARALLELOGRAM
                        POSSREP { A POINT, B POINT,
                                  C POINT, D POINT } } ,
TYPE M IS { PM_PARALLELOGRAM POSSREP { } } ;
```

—and we add both PM_RECTANGLE and PM_RHOMBUS along similar lines to those used for PM_CIRCLE:

```
TYPE PM_PARALLELOGRAM UNION ,
TYPE PARALLELOGRAM IS { PM_PARALLELOGRAM
                        POSSREP { A POINT, B POINT,
                                  C POINT, D POINT } } ,
TYPE PM_RECTANGLE UNION
                  IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RECTANGLE    IS {  PM_RECTANGLE, PARALLELOGRAM
                        POSSREP … } ,
TYPE PM_RHOMBUS UNION
                  IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RHOMBUS    IS {    PM_RHOMBUS, PARALLELOGRAM
                        POSSREP … } ,
TYPE M IS { PM_RECTANGLE, PM_RHOMBUS POSSREP { } } ;
```

The omitted portions, indicated by …, can be found in [1].

Type PM_SQUARE might appear to be more complicated, being an immediate subtype of both PM_RECTANGLE and PM_RHOMBUS, but in fact the definition is simple:

```
TYPE PM_PARALLELOGRAM UNION ,
TYPE PARALLELOGRAM IS { PM_PARALLELOGRAM
                        POSSREP { A POINT, B POINT,
                                  C POINT, D POINT } } ,
TYPE PM_RECTANGLE IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RECTANGLE    IS {  PM_RECTANGLE, PARALLELOGRAM
                        POSSREP … } ,
TYPE PM_RHOMBUS   IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RHOMBUS      IS {  PM_RHOMBUS, PARALLELOGRAM
                        POSSREP … } ,
TYPE PM_SQUARE    IS {  PM_RECTANGLE, PM_RHOMBUS } ,
TYPE SQUARE       IS {  PM_SQUARE, RECTANGLE, RHOMBUS
                        POSSREP … } } ,
TYPE M IS { PM_SQUARE POSSREP { } } ;
```

Figure 2 shows the type graph for this schema.

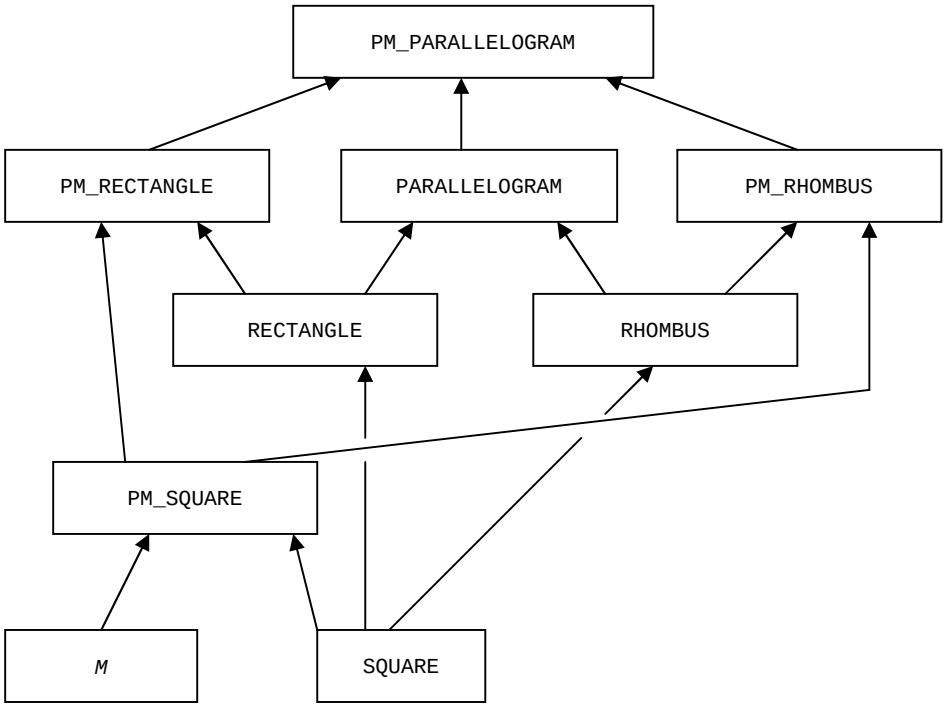

Figure 2: Type graph for `PM_` rectangles and rhombuses

3. Comments

3.1 How many *M* types do we need?

The original idea explicitly proposes that there be "one `M` per root type" (making no assumption concerning the existence of *`alpha`*, which, if supported, is the only scalar root type of the system). Under that proposal, the `M` types of our ellipse and parallelogram examples can be referred to as `M_ELLIPSE` and `M_PARALLELOGRAM`, respectively, as shown in the following very slightly revised definitions:

```
TYPE PM_ELLIPSE UNION ,
TYPE ELLIPSE IS {   PM_ELLIPSE
                    POSSREP { A LENGTH, B LENGTH, CTR POINT } ,
TYPE PM_CIRCLE UNION
               IS { PM_ELLIPSE
                    CONSTRAINT IS_M(PM_ELLIPSE) OR
                               WITH TREAT_AS_ELLIPSE(PM_ELLIPSE) AS E :
                               THE_A(E) = THE_B(E) } ,
TYPE CIRCLE IS {  PM_CIRCLE, ELLIPSE
                  POSSREP { R   = THE_A(ELLIPSE),
                            CTR = THE_CTR(ELLIPSE) } } ,
TYPE PM_O_CIRCLE UNION
                 IS {PM_CIRCLE
                     CONSTRAINT IS_M(PM_CIRCLE) OR
                                THE_R(TREAT_AS_CIRCLE(PM_CIRCLE)) =
                                  POINT(0.0, 0.0) } ,
TYPE O_CIRCLE IS { PM_O_CIRCLE, CIRCLE
                   POSSREP { R = THE_R(CIRCLE) } } ,
TYPE M_ELLIPSE IS { PM_O_CIRCLE POSSREP { } } ;


TYPE PM_PARALLELOGRAM UNION ,
TYPE PARALLELOGRAM IS { PM_PARALLELOGRAM
                        POSSREP { A POINT, B POINT,
                                  C POINT, D POINT } } ,
TYPE PM_RECTANGLE IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RECTANGLE    IS {  PM_RECTANGLE, PARALLELOGRAM
                        POSSREP … } ,
TYPE PM_RHOMBUS   IS {  PM_PARALLELOGRAM
                        CONSTRAINT IS_M(PM_PARALLELOGRAM) OR
                        WITH TREAT_AS_PARALLELOGRAM(PM_PARALLELOGRAM)
                             AS P : … } ,
TYPE RHOMBUS      IS {  PM_RHOMBUS, PARALLELOGRAM
                        POSSREP … } ,
TYPE PM_SQUARE    IS {  PM_RECTANGLE, PM_RHOMBUS } ,
TYPE SQUARE       IS {  PM_SQUARE, RECTANGLE, RHOMBUS
                        POSSREP … } } ,
TYPE M_PARALLELOGRAM IS { PM_SQUARE POSSREP { } } ;
```

A problem now arises when we wish to add a new union type "over" these two existing root types. Consider what might be needed to introduce types `PLANE_FIGURE` and `PM_PLANE_FIGURE`, if `PLANE_FIGURE` is to be a subtype of `PM_PLANE_FIGURE` and a supertype of both `PARALLELOGRAM` and `ELLIPSE`. `PARALLELOGRAM` and `ELLIPSE` have no possreps "in common", so `PLANE_FIGURE` will be another union type (a dummy type, in fact), with `PARALLELOGRAM` and `ELLIPSE` as immediate subtypes:

```
TYPE PM_PLANE_FIGURE UNION ,
TYPE PLANE_FIGURE UNION ,
TYPE PM_PARALLELOGRAM UNION
                        IS { PM_PLANE_FIGURE } ,
TYPE PARALLELOGRAM IS { PM_PARALLELOGRAM, PLANE_FIGURE
                          POSSREP … } ,
… /* rectangles and rhombuses here */
TYPE PM_ELLIPSE UNION
                   IS { PM_PLANE_FIGURE } ,
TYPE ELLIPSE IS { PM_ELLIPSE, PLANE_FIGURE
                    POSSREP … } ,
… (circles here) ,
TYPE M_PLANE_FIGURE IS { PM_SQUARE, PM_O_CIRCLE
                     POSSREP { } } ;
```

This type schema has exactly one *M* type per root type, as proposed. But what has happened to the previously defined M_PARALLELOGRAM and M_ELLIPSE types?

Their existence implies that two distinct values existed in the system, namely, M_PARALLELOGRAM( ) and M_ELLIPSE( ). These distinct values might appear as attribute values in some relvars. The new definition of M_PLANE_FIGURE, however, requires both M_PARALLELOGRAM( ) and M_ELLIPSE( ) to be the very same value: M_PLANE_FIGURE( ). So, introducing type PLANE_FIGURE and its PM_ counterpart would make the existing database, and programs using it, incompatible with the revised type schema. The conversion tasks needed to address this situation are likely to be an unacceptable burden.

There seem to be two ways to cope with this problem:

- Ensure that there is only a single *M( )* value, system-wide (strongly suggesting that this value be defined "close to the *omega* level[5]", so to speak). This will ensure that all existing appearances of *M( )* in the database will be unaffected by the revised type schema.
- Have a separate *M* type for each regular scalar type instead of just for each root scalar type. An immediate consequence is that PM_*T1* is not a proper supertype of PM_*T2* even when *T1* is a proper supertype of *T2*, as can be seen from the following example (irrelevant details omitted):

  ```
  TYPE PM_ELLIPSE UNION,
  TYPE ELLIPSE IS PM_ELLIPSE ...
  TYPE M_ELLIPSE IS PM_ELLIPSE ...

  TYPE PM_CIRCLE UNION,
  TYPE CIRCLE IS { PM_CIRCLE, ELLIPSE } ...
  TYPE M_CIRCLE IS PM_CIRCLE ...
  ```

  Thus, a "missing circle" is no longer the same thing as a "missing ellipse" (and this despite the fact that all circles are ellipses). That is at least somewhat counterintuitive.

[5] The examples already given clearly show that the *M* types necessarily become a leaf type of their PM_ supertypes, thus "pushing them down" towards the *omega* level.

All this strongly suggests that there should be exactly one *M* type in the entire system, at least as far as the scalars are concerned[6]. From now on, therefore, we assume that the *M* type is indeed unique, and is a leaf subtype (ignoring *omega*) of every PM_ type. We will use the name M (no italics) for it.

3.2 Alpha and Omega

IM explicitly defines the concepts of maximal supertypes and minimal subtypes. The maximal supertype of type *T* is the single type that is both a root type and a supertype of *T*. The minimal subtype of type *T* is the single type that is both a leaf type and a subtype of *T*. If *T* is a scalar type, then its maximal supertype is the type, *alpha*, consisting of all scalar values, and its minimal subtype is the empty scalar type, *omega*.

Although *IM* defines *alpha* and *omega*, it does not require them to be available as declared types in an *IM*-conforming language. If *alpha* is not available, then it is still the case that every scalar type has exactly one supertype that is a root type; however, if *omega* is not defined, then a scalar type might have several leaf subtypes.

The questions now arise as to (a) whether the current proposal necessitates the admission, conceptually at least, of types PM_*alpha* and PM_*omega* as well, and (b) if those types are admitted, whether they must be available as declared types in the language.

The proposal expressed in the original e-mail intended the answer to question (a) to be 'no'. The idea was to deal with the notion of 'missing' by using a value for it, and a scalar value in particular. Therefore M( ) had to be considered a scalar value, implying that type M had to be a proper subtype of *alpha* and a proper supertype of *omega*.

Nevertheless, it may be interesting to investigate in some more detail what happens if we do define a type PM_*alpha*, along the lines of the method used before. Doing so would yield:

```
TYPE PM_alpha UNION ;
TYPE M IS { PM_t1, …, PM_tn, POSSREP { } } ;
TYPE alpha UNION ;
```

1. *t1, …, tn* are the names of the regular scalar types that are known to the system. A relational system requires at least one such type (BOOLEAN), so that set of regular types cannot possibly be empty. This implies in particular that whether or not *alpha* is supported does not influence the definition of type M itself (M is always at least a subtype of PM_BOOLEAN), so it (whether or not *alpha* is supported) only determines which union types are available to the user.
2. *alpha* remains the union type for all the regular scalar types. The maximal type for scalar values, however, is now PM_*alpha*.
3. PM_*alpha must* be available as a declared type, because *IM* requires that if two or more types have a common, nonempty subtype, then they must also have a common supertype. However, it does not follow that if PM_*alpha* is available as a declared type, then *alpha*,

[6] Relation and tuple types will be investigated later.

too, must be available. We remark merely that existence of *alpha* might be expected on psychological grounds—if every regular type *t* has a counterpart, PM_*t*, there is an expectation that the existence of type PM_*x* conversely implies that of a type *x*. Moreover, the consequent existence of an IS_*alpha*( ) operator, as a counterpart for IS_M( ), might be much appreciated[7].

Doing a similar exercise for PM_*omega* yields :

```
TYPE PM_omega UNION ;
TYPE M IS { PM_omega POSSREP { } } ;
TYPE omega IS {PM_omega CONSTRAINT FALSE} ;
```

This shows that there is no value of type PM_*omega* that is not also a value of type M. But *IM* states that if *T'* is a proper subtype of *T*, then there must be at least one value of type *T* that is not of type *T'*. Therefore PM_*omega* and M cannot both exist.

3.3 Relation Types and Tuple Types

So far we have defined M in connection with scalar types specifically. Under that assumption, there would be no PM_ counterparts of tuple types and relation types.

That seems undesirable, because the notion of 'missing' can equally well apply to information that, if present, is represented by tuples or relations, and so not supporting the notion of 'missing' for tuple and relation types seems to lead to conceptual incompleteness.

It might be thought that, in the case of relation types, using the empty relation as the value to indicate "missing information" solves the problem. That is incorrect. It may well be the case that an empty relation value is needed to explicitly state something. An empty relation is interpreted as meaning that every instantiation of its predicate is a false proposition, which is very different from saying that there is no relation that can validly appear here.

So it seems that some type like M is needed for use with tuple-valued and relation-valued attributes as well. At first sight, it would then seem natural just to use the same type M and its value M( ) to define PM_TUPLE and PM_RELATION types:

```
TYPE PM_TUPLE { … } UNION ,
TYPE TUPLE { … } IS { PM_TUPLE { … }
                POSSREP ??? } ,
TYPE PM_RELATION { … } UNION ,
TYPE RELATION { … } IS { PM_RELATION { … }
                  POSSREP ??? } ,
TYPE M IS { PM_TUPLE { … }, PM_RELATION { … } POSSREP { } } ;
```

It is not possible to write such declarations in **Tutorial D** because *TTM* requires all relation and tuple types to be system-provided; thus they do not have "declarable" possreps. But this

[7] Without a distinction between *alpha* and PM_*alpha*, finding out whether a value is other than M( ) can be achieved only by using 'NOT IS_M(…)'. Requiring users to write lots of NOTs is not considered a very good idea. The availability of IS_*alpha*(…) might be desirable for that reason alone.

is also the case for system-provided scalar types, such as BOOLEAN, CHAR, and INTEGER. There is nothing to prevent the system from providing PM_ counterparts for all system-defined types.

However, the PM_TUPLE and PM_RELATION types we have now defined consist of a mix of nonscalar values (relation and tuple values, respectively) and a scalar value (M( )). There is one particular aspect of *IM* that might be affected by such a construct, and that is the definition of maximal and minimal types of relation types and tuple types.

Consider a relvar that has an attribute of type RELATION{C CIRCLE}:

```
VAR RVRC RELATION {R_C RELATION{C CIRCLE}};
```

According to *IM*, a corresponding maximal relation type is defined for RELATION {R_C RELATION{C CIRCLE}}, that type being either of the following:[8]

```
RELATION {R_C RELATION{C ELLIPSE}};
RELATION {R_C RELATION{C alpha}};
```

The maximal type is chosen such that all values of RVRC's type are, by definition, also values of that maximal type.

But what would happen if R_C attribute values could be "possibly missing", and we want to reflect that possibility in the design by using a PM_ type ? Our relvar definition would become:

```
VAR PM_RVRC RELATION {R_C PM_RELATION{C CIRCLE}};
```

The maximal relation type can now obviously no longer be any of the two formerly given relation types. However, the corresponding PM_ relation types are suitable candidates:

```
RELATION {R_C PM_RELATION{C ELLIPSE}};
RELATION {R_C PM_RELATION{C alpha}};
RELATION {R_C PM_RELATION{C PM_alpha}};
```
[9]

Now consider

```
RV1 JOIN RV2;
```

where RV1 and RV2 are defined as follows:

```
VAR RV1 RELATION {R_C PM_RELATION{C CIRCLE}};
VAR RV2 RELATION {R_C PM_INTEGER};
```

Note that the R_C attribute in both cases is "possibly missing", meaning in particular that in both relations, M( ) might appear as a value for the R_C attribute. And since there is only one M( ) value under our current assumption, this join has a potentially non-empty result (at least, as far as the rules for join *per se* are concerned)! If we want to assign that non-empty result to

[8] Depending on whether the system has explicit support for type *alpha* (and if not, which type happens to be the root type for type CIRCLE, of course).

[9] Depending on whether PM_*alpha* is indeed defined by the system as discussed previously, and on whether or not PM_*alpha* is indeed the maximal type for scalar types other than M.

a relvar, then we need to be able to find a declared type for the result of the join. *IM* states that this must then be some type

```
RELATION {R_C R_C_TYPE}
```

where *R_C_TYPE* is a supertype of both PM_INT and PM_RELATION {C CIRCLE}. But such a type does not exist! Even *alpha* and PM_*alpha* are only supertypes for the scalars, not for any relation type.

This shows that it is not possible to maintain the idea of one single M type that applies both to scalars and to non-scalars—at least not without drastic revision of *IM*.

We conclude from this that it seems desirable to have, in addition to the type M as defined earlier for the scalars, an M_ relation type generator such that for each relation type there is a corresponding M_ relation type with the same heading, and a similar M_ tuple type generator. Thus, if {*H*} is a heading, then:

- M_RELATION{*H*} is a type having a single value, M_RELATION{*H*}( ), representing, very loosely speaking, a "missing relation" of type RELATION{*H*}. PM_RELATION{*H*} is an immediate supertype of M_RELATION{*H*}.
- M_TUPLE{*H*} is a type having a single value, M_TUPLE{*H*}( ), representing, very loosely speaking, a "missing tuple" of type TUPLE{*H*}. PM_TUPLE{*H*} is an immediate supertype of M_TUPLE{*H*}.

Now, if we replace the PM_ relation type of attribute R_C in RELVAR1 by a scalar one, say PM_CHAR, then a similar reasoning can be applied to demonstrate that support for *alpha* actually becomes a necessity as a consequence of the PM_ types, because PM_*alpha* becomes the maximal type for all PM_ scalars.

We now consider the effects of the foregoing conclusions on the maximal types of relation types and PM_ relation types. Consider first, RELATION { A *st* }, where *st* is an arbitrary scalar type. Clearly this is a subtype of RELATION { A PM_*alpha* }. Equally clearly, that is a subtype of PM_RELATION { A PM_*alpha* }, which is a root type. But PM_RELATION { A PM_*alpha* } is also a supertype of, and therefore the maximal type for, PM_RELATION { A *st* }. So PM_RELATION { A PM_*alpha* } is the maximal type for every unary relation type, and every unary PM_ relation type, whose sole attribute is named A and is of some scalar type. Applying the same argument, *mutatis mutandis*, to tuple types and PM_ tuple types, we conclude that PM_TUPLE { A PM_*alpha* } is the maximal type for every tuple type, and every PM_ tuple type, whose sole attribute is named A and is of some scalar type.

Generalising the foregoing discussion, we find that the maximal type for RELATION { *A1 T1, …, An Tn* }, and also for PM_RELATION { *A1 T1, …, An Tn* }, is PM_RELATION { *A1 MT1, …, An MTn* }, where, for *i* = 1:*n*, *MTi* is the maximal type for *Ti*. For example, PM_RELATION { RVA PM_RELATION { A PM_*alpha* } } is the maximal type for each of the following:

```
RELATION { RVA RELATION { A CHAR } }
```

```
RELATION { RVA PM_RELATION { A CHAR } }
PM_RELATION { RVA RELATION { A CHAR } }
PM_RELATION { RVA PM_RELATION { A CHAR } }
```

This situation may not be desirable—further investigation is needed[10].

3.4 Examples

In this section we present a few examples showing how `PM_` types might be put to good use in relvar declarations, constraints and data manipulation.

In particular, the approach using `PM_` types will be contrasted with the approach proposed in reference [2].

Here are some example relvars, using `PM_` types for some of their attributes:

```
VAR  NATURALPERSON
     RELATION { NR_PERS INTEGER,
                NM_PERS CHAR,
                DT_BIRTH PM_DATE,
                DT_DECEASE PM_DATE,
                MIFID_PROFILE PM_MIFID_PROFILE }
     KEY { NR_PERS } ;

VAR  LEGALPERSON
     RELATION { NR_PERS INTEGER,
                NM_PERS CHAR,
                LEGAL_FORM PM_CHAR,
                DT_FOUND PM_DATE }
     KEY { NR_PERS } ;

VAR  PERSONS_RELATIONSHIPS
     RELATION { NR_PERS_1 INTEGER,
                NR_PERS_2 INTEGER,
                P_RELATIONSHIP_TYPE … }
     KEY { NR_PERS_1, NR_PERS_2 } ;

VAR  CURRENTACCOUNT
     RELATION { NR_ACC CHAR,
                AM_BALANCE AMOUNT }
     KEY { NR_ACC } ;

VAR  SHARES
     RELATION { NR_ACC CHAR }
     KEY { NR_ACC } ;

VAR  PERSONS_CONTRACTS_RELATIONSHIPS
     RELATION { NR_PERS INTEGER,
                NR_ACC CHAR,
                C_RELATIONSHIP_TYPE:… }
     KEY { NR_PERS, NR_ACC } ;
```

[10] There may be similar reasons why it might be desirable to have *alpha*, not `PM_`*alpha*, as the maximal type for the scalar types other than M.

Some remarks on the intended meanings of these relvars:

Persons can be either natural persons, or legal persons. Both are assigned a unique number. Natural persons have a name, a birth date, a dying date, and a mifid profile. Apart from the name, values for each of those attributes are possibly missing (for one reason or another: irrelevant, unknown, etc.).

Legal persons have a name too, a founding date, and a legal form. The legal form is possibly inapplicable (e.g., for companies in a different country).

A person can have a relationship with another person. A natural person can be a legal representative for a legal person or another natural person. A legal person can "own" another legal person, a natural person can be married to another natural person, etc., etc.

Contracts include current accounts and shares accounts (which service stock market orders and suchlike).

Contracts are held by persons (legal or natural), but apart from the actual subscribing persons, other persons can be related to a contract as well—e.g., any other person can be authorised to withdraw funds from a current account by the owner of that account.

Here are some constraints on this database:

1. A shares account must be held by either a legal person or a natural person who has a mifid profile.

```
IS_EMPTY(SHARES JOIN PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = …
         NOT MATCHING (LEGALPERSONS{NR_PERS}
                       UNION
                       (NATURALPERSON WHERE
                                IS_MIFID_PROFILE(MIFID_PROFILE)
                        {NR_PERS})))
```

2. An account cannot be held by a legal person whose foundation date is missing or a natural person whose birth date is missing.

```
IS_EMPTY(PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = … JOIN
         NATURALPERSON WHERE IS_M(DT_BIRTH))
```

   and

```
IS_EMPTY(PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = … JOIN
         LEGALPERSON WHERE IS_M(DT_FOUND))
```

3. A natural person who is the legal representative of another person must be neither a minor nor deceased.

```
WITH (PERSONS_RELATIONSHIPS WHERE P_RELATIONSHIP_TYPE = …) RENAME
NR_PERS_1 AS NR_PERS AS LEGALREPS :
IS_EMPTY(LEGALREPS JOIN NATURALPERSON WHERE (IS_M(DT_BIRTH) OR
DT_BIRTH > NOW() - 18) OR IS_DATE(DT_DECEASE))
```

Without PM_ types, in accordance with reference [2], the design would have to be changed such each attribute of a PM_ type is placed in a relvar of its own to cater for the possible "missingness":

```
VAR  NATURALPERSON
     RELATION { NR_PERS INTEGER,
                NM_PERS:CHAR }
     KEY { NR_PERS } ;

VAR  NATURALPERSON_BIRTHDATE
     RELATION { NR_PERS INTEGER,
                DT_BIRTH DATE }
     KEY { NR_PERS } ;

VAR  NATURALPERSON_DECEASEDATE
     RELATION { NR_PERS INTEGER,
                DT_DECEASE DATE }
     KEY { NR_PERS } ;

VAR  NATURALPERSON_MIFID_PROFILE
     RELATION { NR_PERS INTEGER,
                MIFID_PROFILE MIFID_PROFILE }
     KEY { NR_PERS } ;
```

Extra constraints need to be defined on the above relvars to ensure referential integrity.

```
CONSTRAINT FK1 IS_EMPTY(NATURALPERSON_BIRTHDATE NOT MATCHING
                        NATURALPERSON);

CONSTRAINT FK2 IS_EMPTY(NATURALPERSON_DECEASEDATE NOT MATCHING
                        NATURALPERSON)

CONSTRAINT FK3 IS_EMPTY(NATURALPERSON_MIFID_PROFILE NOT MATCHING
                        NATURALPERSON)
```

Then we would have:

```
VAR  LEGALPERSON
     RELATION { NR_PERS INTEGER
                NM_PERS CHAR }
     KEY { NR_PERS } ;

VAR  LEGALPERSON_LEGALFORM
     RELATION { NR_PERS INTEGER,
                LEGAL_FORM CHAR }
     KEY { NR_PERS } ;

VAR  LEGALPERSON_FOUNDED
     RELATION { NR_PERS INTEGER,
                DT_FOUND DATE }
     KEY { NR_PERS } ;

CONSTRAINT FK4 IS_EMPTY(LEGALPERSON_LEGALFORM NOT MATCHING LEGALPERSON);
```

```
CONSTRAINT FK5 IS_EMPTY(LEGALPERSON_FOUNDED NOT MATCHING LEGALPERSON);
```

The other four relvars, having no PM_-typed attributes, remain unaltered.

The other constraints need to be adapted to deal with the altered structure:

1. A shares account must be held by either a legal person or a natural person who has a mifid profile.

```
IS_EMPTY(SHARES JOIN PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = …
        NOT MATCHING (LEGALPERSONS {NR_PERS} UNION
                      NATURALPERSON_MIFID_PROFILE {NR_PERS}))
```

2. An account cannot be held by a legal person whose foundation date is missing or a natural person whose birth date is missing.

```
IS_EMPTY(PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = … JOIN
       (NATURALPERSON NOT MATCHING NATURALPERSON_BIRTHDATE))
```

   and

```
IS_EMPTY(PERSONS_CONTRACTS WHERE C_RELATIONSHIP_TYPE = … JOIN
       (LEGALPERSON NOT MATCHING LEGALPERSON_FOUNDED)
```

3. A natural person who is the legal representative of another person must be neither a minor nor deceased.

```
WITH (PERSONS_RELATIONSHIPS WHERE P_RELATIONSHIP_TYPE = …) RENAME
NR_PERS_1 AS NR_PERS AS LEGALREPS :
IS_EMPTY(LEGALREPS
   JOIN (((NATURALPERSON NOT MATCHING
           NATURALPERSON_BIRTHDATE) {NR_PERS})
   UNION ((NATURALPERSON NOT MATCHING
           NATURALPERSON_DECEASEDATE) {NR_PERS})
   UNION ((NATURALPERSON_BIRTHDATE
           WHERE (DT_BIRTH > NOW()-18)){NR_PERS})
        )
       )
```

Manipulation of these relvars in either design is pretty straightforward. In each example we give the solution for the design using PM_ types followed by the solution using decomposition.

- Register a new natural person whose birth date is known:

```
INSERT NATURALPERSON RELATION{TUPLE{NR_PERS …, NM_PERS …, DT_BIRTH …,
DT_DECEASE M( ), MIFID_PROFILE M( )}};
```

```
INSERT NATURALPERSON RELATION{TUPLE{NR_PERS …, NM_PERS … }}, INSERT
NATURALPERSON_BIRTHDATE RELATION{TUPLE{NR_PERS …, DT_BIRTH … }};
```

- Remove a natural person from the database:

```
DELETE NATURALPERSON WHERE …;
```

```
DELETE NATURALPERSON WHERE …, DELETE NATURALPERSON_BIRTHDATE WHERE …,
DELETE NATURALPERSON_DECEASEDATE WHERE …, DELETE
NATURALPERSON_MIFID_PROFILE WHERE …;
```

- Set an inappropriately entered decease date back to "missing":

  ```
  UPDATE NATURALPERSON WHERE … (DT_DECEASE := M( ));
  DELETE NATURALPERSON_DECEASEDATE WHERE …;
  ```

- Get the highest age of all natural persons whose age is known:

  ```
  NOW() - MIN(NATURALPERSONS : IS_DATE(DT_BIRTH), DT_BIRTH)
  NOW() - MIN(NATURALPERSONS_BIRTHDATE, DT_BIRTH )
  ```

- Get the count of natural persons that do not have a mifid profile:

  ```
  COUNT (NATURALPERSONS WHERE IS_M(MIFID_PROFILE))
  COUNT (NATURALPERSONS NOT MATCHING NATURALPERSONS_MIFID_PROFILE)
  ```

REFERENCES

1. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Boston, Mass.: Addison-Wesley (2006).
2. Hugh Darwen: *How to Handle Missing Information Without Using NULL*. Annotated lecture slides available at http://www.thethirdmanifesto.com.

End of paper