Towards an Agreeable Model of Type Inheritance

or

Object Identifiers and Inheritance Don't Mix!

Hugh Darwen

Abstract— We (the authors of [3]) contend that, to be agreeable, a model of type inheritance must embrace the concept of specialization by constraint. I describe how we came to this conclusion and explain how we think a truly relational system can incorporate such a model, by avoiding the use of object identifiers.

Index Terms—Inheritance, Types, Relational Model, Object Identifiers

1 INTRODUCTION

Chapter 3, *The Third Manifesto*, of [3] is our proposed restatement of the Relational Model of Data ([2]). It includes the following point in its final section, headed "OO VERY STRONG SUGGESTIONS":

1. Some form of **type inheritance** should be supported (in which case, see OO Prescriptions 2 and 3). In keeping with this suggestion, **D** should not support **coercions** (i.e., implicit type conversions).

The two referenced "OO Prescriptions" are given a little earlier in the same chapter:

2. If **D** permits some type *T′* to be defined as a **subtype** of some **supertype** *T*, then such a capability shall be in accordance with some clearly defined and generally agreed model.
3. If **D** permits some type *T′* to be defined as a subtype of some other type *T*, then *T′* shall not be prevented from additionally being defined as a subtype of some other type that is neither *T* nor any supertype of *T* (unless the requirements of OO Prescription 2 preclude such a possibility).

Taken out of context as they are here, these points need some explanation. "**D**" is a generic name for any database language that conforms to our model. "Prescriptions" are mandatory requirements. They are subdivided into RM Prescriptions, being those derived from the Relational Model, and OO Prescriptions, where OO stands for Other Orthogonal and coincidentally includes prescriptions derived from object orientation. Very Strong Suggestions, also subdivided into RM and OO sections, are points that we think are good ideas but not essential for conformance. The word Very is included to emphasize that such sections are not intended to provide a compendium of *all* relevant ideas we think are good ones.

It should now be clear that we think it highly desirable, though not essential, for a relational DBMS to support type inheritance (point 1), but only if it does so in accordance with a generally agreed model (point 2) that includes what is commonly called "multiple inheritance" (point 3).

Our restatement of the Relational Model has the following aims, among others:

- To clarify or remove certain points about the Relational Model that we feel were not clear in 1970 and have remained unclear since then. For example, we no longer have any reference to the unclear concept of *atomicity* of values that was part of Codd's definition of First Normal Form of relations.
- To elaborate on the "domain" aspect of Codd's model, in a theory of types formulated as rigorously as the theory of relations.

These two aims in particular, we thought, justified the occurrence of the term "Object/Relational" in the book's title, for support for types of arbitrary complexity and support for type inheritance were the two features commonly found in object oriented languages that we strongly wished to include. We were running with the tide, of course, because those two features now constitute perhaps the biggest distinction between SQL:1999 ([6]) and its predecessor, SQL:1992 ([5]); and of course they feature strongly in the ODMG standard ([1]). Indeed, SQL:1999 is sometimes seen as SQL with added "object" features, while the ODMG standard might be considered to be about object oriented databases with added "relational" features (i.e., OQL).

Early published versions of *The Third Manifesto* elaborated a little on that "generally agreed model" in point 2, as follows:

> It is our hope that such a "clearly defined and generally agreed model" will indeed someday be found. The term "generally agreed" is intended to imply that the authors of this *Manifesto*, among others, shall be in support of the model in question. Such support shall not be unreasonably withheld.

Because these remarks, not intended to be taken too seriously, were much misunderstood, we decided not to include them in the book. However, they were based on an observation that *was* generally agreed at the time, namely, that no generally agreed model of type inheritance existed.

Rather boldly, perhaps, we decided to attempt to formulate a model of type inheritance, with the idea of presenting it as a "strawman proposal" for the generally agreed one being sought. Our model made its first appearance in the form of some preliminary proposals constituting an appendix in an early version of *The Third Manifesto.* By the time the book ([3]) was being prepared for publication, we decided that our model had matured sufficiently to appear as Part IV, Subtyping and Inheritance, in the main body of the book.

However, when we reflected on our completed work—intended as a rigorous definition of a certain model of type inheritance—we decided that it did *not* meet that originally published criterion of acceptability to "the authors of this *Manifesto*"! Our reasons for rejecting our own proposal are given in Appendix C of the book, which is titled, highly significantly, **Specialization by Constraint**.

Appendix C starts to point to a possible way of enhancing our model to make it acceptable to us. We are currently working on a revision and are cautiously optimistic about it.

The purpose of this informal article is to explain:

- some of the basics that are common to our original model and the revision that is in progress;
- what drove us to formulate a rigorous model that we would eventually have to reject;
- why we rejected it;
- the salient features of what will be our revised model;
- why we think SQL:1999, Java™ and ODMG do not support this model.

2 BASICS

Before you can have type inheritance, you need types. In order to have whatever types are desired, you need not only a judiciously chosen set of built-in types, but also a facility to allow users to define additional types of arbitrary complexity.

A type is a defined finite set of values and associated operators. The associated operators consist of those defined to operate on values or variables of that type and those that, when invoked, return values of that type.

A type is defined by a named *possible representation* (*possrep* for short) and a *type constraint.* We stress that there is no prescribed relationship between possible representations (defined in the model) and actual representations (defined in the implementation).

A possrep consists of one or more named components, each having, as well as a name, a *declared type*.

For example, suppose type POINT consists of values representing points in the Euclidean plane. Then the possrep XY_POINT might be defined thus:

```
POSSREP XY_POINT { X NUMERIC,
                   Y NUMERIC }
```

where NUMERIC is some previously defined (possibly built-in) type. A particular POINT value can therefore be considered to consist of an X component and a Y component, both numbers.

A possrep definition implies the existence of certain operators. These operators are our proposed counterparts of what are sometimes referred to (e.g., in SQL:1999) as *observers, mutators* and *constructors* (terms we do not use, partly because they have different meanings in, for example SQL:1999 and Java™).

Analogous to SQL:1999 "observer methods" are the operators, implied by XY_POINT, called THE_X and THE_Y.[1] Given a value, *p,* of type POINT, the invocations THE_X (*p*) and THE_Y (*p*) return the X and Y coordinate, respectively, of *p*.

Our counterpart of Java™ "constructor functions" is a class of operators we call *selectors*. The selector implied by XY_POINT is the operator of that name with two parameters, both of declared type NUMERIC, corresponding to the two components of XY_POINT. Thus, the invocation XY_POINT (0, 0) returns the point that is the origin. We call this operator the XY_POINT selector for values of type POINT.

THE_X, THE_Y and XY_POINT are examples of what we call *read-only operators*. A read-only operator is one that, when invoked, operates on zero or more given values (arguments) and returns a value.

In contrast to read-only operators, we have *update operators*. An update operator, when invoked, operates on at least one *variable* and zero or more values, and does not return anything.

The definition of XY_POINT implies the existence of two update operators. One of these, given a variable, *pv*, of declared type POINT and a number, *x*, assigns the value XY_POINT (*x*, THE_Y (*pv*)) to *pv*; the other, given the same variable and a number, *y*, assigns the value XY_POINT (THE_X (*pv*), *y*) to *pv*. Rather than give specific names for these operators, we treat them merely as special forms of assignment.

The type constraint which, in combination with possrep XY_POINT, defines POINT, is merely *true*.[2] From this it can be seen that if *l* and *h* are the lowest and highest values of type NUMERIC, then every point within a certain square of side *h--l* is representable and, indeed, the set of such values constitutes the type POINT. This brings me to an important aspect of our model: given a possrep *P* and a constraint *C*, for type *T*, then every value in type *T* can be expressed by some invocation of the selector *S* implied by *P*, and every invocation of *S* that satisfies *C* returns some value in *T*. It follows that if there is such a thing as a colored point (to cite an example the like of which is commonly found in object oriented literature), consisting of a point and a color, then that thing is not a value of type POINT. In other words, our model puts a big question mark on the concept of defining a subtype by extension of representations defined for the supertype (i.e., "adding an attribute").

In case the reader is wondering, we do not insist on exactly one possrep/constraint pair per type definition, though the point is not germane to this article.

Our model requires, in addition to the operators implied by possrep definitions, several further operators to be available in connection with every type—in particular, "equals" comparison and assignment. We require that if $x = y$, then x and y are indistinguishable—really are the same value—implying that for every read-only operator f defined for values of the type of x and y, $f(...x...) = f(...y...)$ is *true*. We further require the effect of assignment of a value v to a target t to have the effect that $t = v$ is subsequently *true*.

3 MOTIVATION AND BASIS FOR A MODEL OF TYPE INHERITANCE

Quite simply, our motivation was that something called type inheritance was commonly deemed to be a characteristic feature of object-oriented systems and was much talked about as a strongly desired addition to relational systems. And yet,

[1] All concrete syntax is offered for purposes of illustration and exposition only. We do not prescribe syntax to be used in implementations.

[2] An example of a type requiring a non-trivial constraint might be ANGLE, being represented by numbers in the range 0 to 2π

time and again we encountered articles in the literature bemoaning the lack of a commonly agreed model, and even the lack of agreement on what is meant by a statement of the form "every *y* **is an** *x*".

We decided very early on that to say that type *T2* is a subtype of type *T1* is to assert that every value in *T2* **is a** value in *T1*. We further decided that a crystal-clear example is that of CIRCLE, being a subtype of ELLIPSE, meaning that every value of type CIRCLE is a value of type ELLIPSE or, in everyday parlance, every circle is an ellipse, though not every ellipse is a circle. We say that CIRCLE is a *more specific* type than ELLIPSE, and that if a value *e1* has, for example, types CIRCLE, ELLIPSE, PLANE_FIGURE (being a supertype of ELLIPSE) and no other, then CIRCLE is the *most specific type* of *e1*. The most specific type of a value having only the types ELLIPSE and PLANE_FIGURE is ELLIPSE.

Because the truth of that statement about circles and ellipses is, like other similar statements concerning geometrical plane figures, so crystal-clear to us, we very deliberately chose to develop our model around the study of such examples, rather than certain examples of a subtly different kind that we found in much of the object oriented literature. I am referring here to examples such as MANAGER being a subtype of EMPLOYEE, TOLL_HIGHWAY of HIGHWAY and COLOURED_CIRCLE of CIRCLE, where a manager is an employee with a budget, a toll highway a highway with tolls, a colored circle a circle that has a color; employees in general don't have budgets, highways in general don't have tolls and circles in general don't have colors. We did not at the outset have any intention to outlaw such examples; we merely regarded them as "fuzzy" in comparison with our crystal-clear spatial ones and we wanted to avoid any possibility of being beguiled by the fuzziness into unwise decisions regarding the definition of our model.

To complete our basis, we had to decide what important consequences would arise from CIRCLE being a subtype of ELLIPSE. We studied [8] and discussed this text with several people. In connection with type inheritance, the authors of [8] posit four desiderata: substitutability, static type checking, mutability, and "specialization via constraints". They conjecture that it is not possible to build a computer system that embraces all four of these concepts, though it *is* possible to build one that embraces any three of them. We were very interested in this conjecture, for we certainly did not want to propose a model that could not be implemented!

To understand the conjecture, we first of all had to understand very precisely the four concepts in question—for we imagined that we would have to decide which one to exclude from *our* model. Acquiring this understanding turned out to be remarkably difficult, which is one reason why [3] includes six pages of discussion under the heading THE "3 OUT OF 4" RULE. I will now discuss each of the cited desiderata and present our conclusion in each case as to whether it should belong in our model.

4 SUBSTITUTABILITY

Substitutability is sometimes expressed in terms of "instances". For example, you might find in the literature a statement such as this: If *T2* is a subtype of *T1*, then it is the case that everywhere an instance of *T1* is expected, an instance of *T2* is permitted. But our model has to be a possible basis for a computer language. In such a language, the "instances" in question are represented by expressions denoting operands. Some expressions denote *values*; others denote *variables*. This particular distinction perhaps seems trite, but very early on we were struck by (a) its fundamental importance and (b) an apparent failure to observe it, by some of those who like to refer just to "instances".

We therefore discussed two distinct concepts: *value substitutability* and *variable substitutability.* We quickly realized that value substitutability is essential. For example, let AREA be an operator defined for ellipses such that if "E" is an expression denoting some ellipse, then the expression "AREA (E)" is an expression denoting the value of type AREA that is the area of E. Now, if "C" is an expression denoting a circle, then the expression "AREA (C)" is implicitly valid *and denotes the area of C.* My emphasis is to indicate that we attach importance to the concept that the operator on ellipses is not only "inherited" by circles, but also has the same meaning for circles as it does for ellipses.

Value substitutability applies not only to invocations of read-only operators, such as AREA, but also to those parameters of an update operator that are not subject to update. To clarify what I mean here, consider ordinary assignment, such as "X := Y + 2". I consider the ":=" operator to have two parameters, which we might refer to as the *source* (Y + 2 in my example) and the *target* (X in the example). The target is subject to update, whereas the source is not. Value substitutability clearly applies to the source. If E is a variable of type ELLIPSE and "C" is an expression denoting a circle, then "E := C" is a valid assignment. The question now arises as to whether substitutability applies to parameters that are subject to update.

An argument substituted for a parameter that is subject to update must be a variable. To decide whether *variable substitutability* was a concept to be embraced, we considered the simplest of all update operators, namely, assignment involving a single target, that being denoted by a simple variable name. It then became transparently obvious that variable substitutability does not in general make good sense. For consider:

```
(1)  TYPE ELLIPSE POSSREP (  A LENGTH,
                             B LENGTH,
                             C POINT );

(2)  TYPE CIRCLE POSSREP ( R LENGTH,
                           C POINT )
           SUBTYPE_OF ( ELLIPSE ) ;

(3)  VAR E ELLIPSE ;

(4)  VAR C CIRCLE ;

(5)  E := ELLIPSE ( LENGTH ( 5 ),
                    LENGTH ( 4 ),
                          XY_POINT ( 0, 0 )
                          ) ;
(6)  C := CIRCLE ( LENGTH ( 5 ),
                   XY_POINT ( 0, 0 )
                   ) ;
(7)  E := CIRCLE ( LENGTH ( 5 ),
                   XY_POINT ( 0, 0 )
                   ) ;
(8)  C := ELLIPSE ( LENGTH ( 5 ),
                    LENGTH ( 4 ),
                    XY_POINT ( 0, 0 )
                    ) ;
```

(1) defines a type called ELLIPSE and (2) defines CIRCLE as a subtype of ELLIPSE. The constraints for these types are both *true* by default.

The possrep given for ELLIPSE consists of components representing the major semiaxis (A), the minor semiaxis (B) and the center (C). Because no name is explicitly given for this possrep, its name is by default the type name, ELLIPSE. For simplicity we assume that all the ellipses we want to "talk about" are ones whose major axis is parallel to the X axis.

The possrep given for CIRCLE consists of components representing the radius (R) and the center (C). The implicitly defined name of this possrep is CIRCLE.

(3) declares a variable, E, to be of type ELLIPSE (we say that ELLIPSE is the *declared type* of E) and (4) declares a variable, C, to be of type CIRCLE.

(5) is permitted and assigns a certain ellipse to E. (6) is likewise permitted and assigns a certain circle to C.

(7) is permitted under value substitutability and assigns a certain ellipse (in fact, a certain circle) to E.

(8) is not permitted. It is attempting to assign to a circle variable a value that is not a circle. Such an attempt is an example of a *type error*.

That (8) is a type error is common to many languages we are aware of that support the concept of type inheritance, including those specified in SQL:1999, ODMG and Java™. We concluded that it cannot in general be the case that if *T2* is a subtype of *T1*, then wherever a variable of declared type *T1* is expected, a variable of declared type *T2* is permitted. Looking at simple assignment, it might be thought that the inverse is the case: wherever a variable of declared type *T2* is expected, a variable of declared type *T1* is permitted. But this is not so in general, either. For consider:

```
(9)      C.R := LENGTH ( 6 ) ;
(10)     E.R := LENGTH ( 6 ) ;
```

(9) is of a form commonly found in object oriented languages. It is a realization of invoking the update operator implied by the declaration of the R component of CIRCLE's possrep. This operator, recall, given a variable *cv* of declared type CIRCLE and a length *l*, has the effect of assigning CIRCLE (*l*, THE_C (*cv*)) to *cv*.

(10) is not permitted. The ELLIPSE possrep doesn't have an R component.

In view of this observation, we discarded any notion that a concept of variable substitutability is generally applicable.

But typical object oriented languages do seem to support variable substitutability of a sort. Consider the following further assignments:

```
(11)     E.A := LENGTH ( 6 ) ;
(12)     C.A := LENGTH ( 6 ) ;
```

The effect of (11) is, loosely speaking, to update the A component of the value assigned to E.

(12) is permitted in SQL:1999, ODMG and Java™ (henceforth referred to collectively as SQL:1999 *et cetera*]), as a consequence of the legality of (11). Contrast this with the illegality of (8) in SQL:1999 *et cetera*. According to our model, (12) is *not permitted* because there is no possrep in the type definition of CIRCLE that has a component named A. However, the read-only operator, THE_A, implied by the A component of the possrep given in the type definition of ELLIPSE, most definitely *is* inherited by circles, as is every read-only operator that is defined for ellipses[3] in general.

We easily justify this difference between the [3] model and the implementations found in SQL:1999 *et cetera*.[4] To update the A semiaxis of a circle variable, even if circle variables are considered to have such a component, is very likely to result in the variable in question being assigned a value that is not a circle. We further observe that in SQL:1999 *et cetera*, it is indeed possible for (12) to have the effect of assigning to C a value such that THE_A (C) ≠ THE_B (C) (and who knows what the value of THE_R (C) might then be?). We are well aware that it is commonly observed that SQL:1999 *et cetera* are just not suitable for the "ellipses and circles" example; rather, they are much better suited to the "employees and managers" example. But we want a language that is suited to crystal-clear cases, even if to have such a language is to sacrifice support for the fuzzy cases of subtyping.

Our conclusion regarding substitutability is that without value substitutability a system simply cannot be said to be supporting type inheritance in any agreeable way. Unconditional variable substitutability, however, even in the restricted sense in which it is found in SQL:1999 *et cetera*, seems not to make much sense.

5. STATIC TYPE CHECKING

Static type checking refers to the property of a language whereby all type checking occurs at "compile time", meaning that all type errors can be discovered merely by inspection of program text. If static type checking is fully supported, possibly expensive run-time checks are not needed. Further, applications are less likely to fail at run-time than they might otherwise be.

[3] To be precise, we mean every read-only operator that has a parameter whose declared type is some supertype of ELLIPSE (possibly ELLIPSE itself).

[4] These implementations do not appear to be based on any clearly defined model.

We think that static type checking is a strong desideratum for industry-strength database languages. We have encountered much support for this position in the database community at large, and no significant opposition to it.

Our model is very similar to SQL:1999's with respect to static type checking. We have static type checking except in one specific place in the language, which I will now explain.

Consider again statement (7), in which ellipse variable E is assigned a circle. Under certain circumstances we might want to compute the radius of this circle. However, the following expression "THE_R (E)" is *not permitted*, as THE_R is not defined for ellipses in general. To get around this problem we advocate the provision of a generic operator which we call TREAT_DOWN_AS_*T*, where *T* is a type name. (SQL:1999 has a similar operator, called TREAT.) For example, the following expression is permitted:

```
(13) THE_R ( TREAT_DOWN_AS_CIRCLE ( E ))
```

When the value of the operand of TREAT_DOWN_AS_CIRCLE happens to have type CIRCLE, then the result of the invocation is that value; otherwise the result of the invocation is an exception—a run-time type error. Because every successful evaluation of TREAT_DOWN_AS_CIRCLE (E) is guaranteed to yield a circle, the declared type of that expression is CIRCLE and so the expression is permitted as an argument to an invocation of THE_R.

In our view, the provision of TREAT_DOWN_AS_*T* is an inevitable consequence of substitutability alone. Even though it can cause run-time type errors, it does so in a controlled manner that allows applications to defend themselves against those errors as easily as they can against run-time errors in general. I will later describe an intolerable kind of run-time type error that definitely cannot occur in our model.

6. MUTABILITY

This desideratum, as far as we can see, refers to nothing more than what we call update operators—primarily, assignment. We have no reason to be opposed in principle to functional programming languages, which manage without variables and assignment, but we felt that departure from the imperative style would have created too much of a diversion from our main purpose. We do, therefore, support operators that update the database, though, like [2], we restrict such operators to those that update *relation variables*. This restriction does not apply to operators for use on local variables in application programs.

7. SPECIALIZATION BY CONSTRAINT

When we encountered this concept, we took it to refer to the idea that the most specific type of the value assigned to a variable might change as the result of some update operation on that variable. When the most specific type changes from ELLIPSE to CIRCLE, we have a case of specialization. When it changes from CIRCLE to ELLIPSE, we have *generalization*, so the concept we are discussing really concerns both specialization and generalization and the term we have chosen for it is not quite as apt as we would like.

Suppose that variable E is currently assigned a value whose most specific type is ELLIPSE. Clearly, assigning to E the result of invoking the CIRCLE selector would now cause the most specific type of E to change to something more specific than ELLIPSE. Under the principle of value substitutability, the most specific type of a variable *can* change from time to time. The question is, can it change as the result of an assignment of an expression that does not explicitly specify a value of some proper subtype of ELLIPSE? For example, consider this assignment:

```
(14)  E := ELLIPSE ( LENGTH ( 5 ),
                     LENGTH ( 5 ),
                     XY_POINT ( 0, 0 )
                     ) ;
```

The declared type of the source of this assignment is clearly ELLIPSE, and by definition ELLIPSE is also the most specific type of the value yielded by an invocation of the ELLIPSE selector. Under specialization by constraint, we thought, the most specific type of this result would nevertheless be CIRCLE (more precisely, some subtype of CIRCLE, possibly CIRCLE itself), assuming that type CIRCLE is somehow defined by a constraint meaning that *C* is a circle *if and only if* THE_A (*C*) = THE_B (*C*). A careful reading of [8], however, reveals that the authors there were referring merely to what we would call *type constraint enforcement*, under which if *C* is a circle, then THE_A (*C*) = THE_B (*C*), but if THE_A (*C*) = THE_B (*C*), then *C* is not necessarily a circle. We prefer our interpretation of the phrase.

Specialization by constraint appeals so strongly to our normal intuition as to make it apparently a *sine qua non*. Human beings understand this concept so well[5] that, surely, we thought, a model of type inheritance should be judged by the extent to which it embraces it.

Now, [8] gives a certain example, to which the following is isomorphic, assuming statements (1) to (6) to be in effect:

```
(15)     E := C ;

(16)     E.A := LENGTH ( 6 ) ;
```

The authors claim that if specialization by constraint is in effect, then (16) gives a run-time type error. This is because the most specific type of E is CIRCLE, but the assignment in (16) would cause it to acquire a value that is inconsistent with being a circle.

Now, the reader might well be a little puzzled at this stage, as we were when we first encountered [8] and as we remained even after a considerable amount of face to face discussion with various experts. Why, we might ask, isn't (16) equivalent to "E := ELLIPSE (LENGTH (6), THE_B (E), THE_C (E)), as in (11)? Be that as it may, to cut a long story short, we decided to go with the field, so to speak. Like

[5] Perhaps I exaggerate. Some maintain that a square isn't a rectangle, because increasing the width of a rectangle yields another rectangle, whereas increasing the width of a square doesn't yield another square (but rather, I would add, yields another rectangle).

SQL:1999 *et cetera*, we decided to reject specialization by constraint. The most important consequence of this decision, to us, was that we were eventually to realize that the model of type inheritance we had proposed in [3] was definitely not an agreeable one, to us.

It now becomes necessary for me to distinguish between our first model, that we now reject, and the model we will propose in our forthcoming second edition of [3][6]. I will refer to these two models as Model 1 and Model 2. As the reader might have guessed by now, the most important distinguishing feature between Model 1 and Model 2 is that Model 2 embraces specialization by constraint, whereas Model 1 does not.

8. FEATURES OF MODEL 1

As I have already indicated, Model 1 has value substitutability, static type checking that does not entirely eliminate run-time type errors, and mutability. It does not have specialization by constraint but, unlike SQL:1999 *et cetera*, it does have type constraint enforcement.

One small but crucial point that our model has in common with SQL:1999 *et cetera* is that although a value can be of more than one type, every value has exactly one *most specific type*. To recap and rephrase slightly, a most specific type of a value *v* is a type such that no proper subtype of it is a type of *v*. (Every type is a subtype of itself. For that reason, we need the term "proper subtype" to refer to a type *T* that is a subtype of *T* other than *T* itself.) Further, we allow the most specific type of a value to be a type that has one or more proper subtypes (i.e., a type that is not a *leaf type*).

The following features of Model 1 are important but not especially germane to the present discussion:

- *Multiple inheritance.* A type can have more than one immediate supertype. For example, SQUARE might be a subtype of both RECTANGLE and RHOMBUS, neither of which is a supertype of the other (but both of which are possibly subtypes of PARALLELOGRAM). We did not turn our minds to multiple inheritance until we felt we had completely nailed down all the details of single inheritance. We then found, contrary to experiences reported orally to us by several other investigators, that multiple inheritance presented no significant extra difficulty.
- *Tuple type inheritance.* Because [3] embraces the Relational Model, it prescribes the provision of *type generators* (as we call them—they are also variously referred to as type constructors, parameterized types and type templates) for tuple and relation types. We therefore had to consider how type inheritance would apply to tuple types and relation types. It was manifestly clear to us that if CIRCLE is a subtype of ELLIPSE, then TUPLE { E CIRCLE, C COLOUR } is a subtype of TUPLE { E ELLIPSE, C COLOUR }. Embracing this concept presented no significant difficulty. It did have the interesting side effect of proving that, in relational systems at least, the provision of support for single inheritance implies the provision of support for multiple inheritance! The types TUPLE { E1 CIRCLE, E2 ELLIPSE } and TUPLE { E1 ELLIPSE, E2 CIRCLE } are manifestly both supertypes of TUPLE { E1 CIRCLE, E2 CIRCLE }; equally manifestly, neither is a supertype of the other.
- *Relation type inheritance.* For every tuple type there is a corresponding relation type having the same attribute definitions. Clearly, if tuple type *TT2* is a subtype of tuple type *TT1*, then the relation type having the same attribute definitions as *TT2* is a subtype of the relation type having the same attribute definitions as *TT1*. This observation did raise some nontrivial questions. For example, consider unary relations *r1* and *r2* such that declared type of the only attribute, E, of *r1* is ELLIPSE, while that of the only attribute, E, of *r2* is CIRCLE. What is the declared type of *r1* JOIN *r2*? At first glance, it appears to be RELATION { E CIRCLE }, for it can be seen that for every possibly combination of values for *r1* and *r2*, in every tuple of the result the value for E must be a circle. However, we show in [3] that the declared type of *r1* JOIN *r2* has to be RELATION { E ELLIPSE }. The reasoning that leads to this conclusion is not difficult but is beyond the scope of this article. We note that SQL:1999 comes to a similar conclusion.

  We wondered if the most specific type of a relation might be a proper subtype of its declared type. For example, consider again a unary relation *r1* whose only attribute, E, is of declared type ELLIPSE. Might the most specific type of *r1* be some proper subtype of RELATION { E ELLIPSE }? We decided that if in every tuple of *r1* E is in fact a circle, then the most specific type of *r1* must be no less specific than RELATION { E CIRCLE }. Suddenly the specter of the empty relation loomed! What is the most specific type of *r1* if *r1* has no tuples at all? Again, [3] has an answer and again the answer is beyond the scope of the present article.

A feature of Model 1 that most definitely *is* germane to the present discussion is the generic operator we call TREAT_UP_AS_*T* (I have already described TREAT_DOWN_AS_*T*).

Consider again statement (14), whose effect is to assign to the variable E an ellipse with semiaxes of equal length. Also consider again statement (6), whose effect is to assign to the variable C a circle whose radius is of the same length as the semiaxes of E and whose center is the center of the ellipse assigned to E. Assume these two statements to be in effect. The comparison "E = C" is permitted in Model 1 and returns *false*. To provide for comparison to determine if the value of C really is the same ellipse (intuitively) as the value of E, we have to introduce a certain artifice, and TREAT_UP_AS_*T* is

[6] Incidentally, we are proposing to take the unusual step of changing the title, in this second edition, to "Foundation for Future Database Systems: *The Third Manifesto*".

that artifice. For example, the following expression returns *true* under the given circumstances:

```
E = TREAT_UP_AS_ELLIPSE ( C )[7]
```

being equivalent to

```
E = ELLIPSE ( THE_R ( C ),
              THE_R ( C ),
              THE_C ( C ) )
```

The expression

```
TREAT_DOWN_AS_CIRCLE ( E ) = C
```

also returns *true*, but this latter expression might in other circumstances result in a run-time type error, which the expression using TREAT_UP_AS_ELLIPSE is guaranteed not to.

Now consider again statements (15) and (16). (15) is permitted in Model 1, as already explained as a consequence of substitutability. (16), however, might well not be permitted, while (15) is in effect, though not exactly for the reason given in [8]. For (16) to be permitted in Model 1, the update operator defined for ellipses, that has the effect depicted in (16) as assignment to E.A (not, I hasten to add, a shorthand that we use in our examples in [3]), would have to be defined also for circles. Recall that subtypes do not in general inherit update operators from their supertypes. Without going into details, I will just say here that TREAT_UP_AS_*T* again comes to the rescue here, but it leads to the possibly surprising necessity to write TREAT_UP_AS_ELLIPSE (E) as part of the target of an assignment. In case the reader isn't immediately surprised, note that the declared type of E is ELLIPSE and yet we apparently have to tell the system that we wish its current value to be treated as an ellipse. What we are really saying is that we wish it to be treated as an ellipse, indeed, but as nothing more special than an ellipse, even if that current value happens to be, for example, a circle.

We consider the necessity for TREAT_UP_AS_*T* to be a defect in Model 1. We also consider the effect of statement (14) in Model 1 to be a defect—the value being assigned to E here most definitely *is a* circle, which the system should be able see as clearly as any human being can. We consider these defects to militate against acceptance of Model 1 as an agreeable model of type inheritance. It follows that we also reject the type inheritance mechanisms defined in SQL:1999 *et cetera*.

9. FEATURES OF MODEL 2

Model 2 is effectively Model 1 plus specialization by constraint. In gaining specialization by constraint it loses those features of Model 1 that we found obnoxious.

The generic TREAT_UP_AS_*T* operator has gone away altogether.

[7] Actually, if it is possible that the value currently assigned to E is of some proper subtype of ELLIPSE, TREAT_UP_AS_ELLIPSE would be needed on both comparands: TREAT_UP_AS_ELLIPSE (E) = TREAT_UP_AS ELLIPSE (C).

Value substitutability and static type checking are retained and, in fact, improved. In particular, statement (15) no longer has the effect of making (16) cause a run-time type error, even though we retain the fact that update operators that operate on variables of the supertype are not necessarily inherited by variables of the subtype. Note that E is a variable of type ELLIPSE, not a variable of type CIRCLE.

At the present level of discussion, that is all that needs to be said about Model 2. I hope, though, that the question immediately arises in the reader's mind as to why (on earth!), in that case, we didn't go for Model 2 in the first place. The answer, quite simply, is that we were given to believe it was not feasible. On reflection, *we have changed our mind* about that (and are at odds with [7]).

I now want to explain why we think Model 2 is, after all, feasible, and why we believe such a model is not supported in SQL:1999 *et cetera*.

10. WHY MODEL 2 IS FEASIBLE

We consider the feasibility of much of Model 2 to have been already demonstrated in implementations to date of SQL:1999 *et cetera*. Take what is commonly known as "run-time binding", for example. It is true that we are slightly more demanding in this respect, than are SQL:1999 *et cetera*, for we do not accept the concept known as "the distinguished parameter" (i.e., binding based only on the most specific type of the first argument, or "the object to which the message is sent", as Smalltalk users would have it). Rather, we advocate the approach of what we have seen referred to as "multifunctions", requiring the matching of the most specific types of all of the arguments to the declared types of corresponding parameters. Early drafts of SQL:1999 had this feature, even though it was ultimately removed. I am reliably informed that at least one well known DBMS vendor has already implemented it.

Our concept of possible representations might at first sight appear to be novel and require proof of feasibility. SQL:1999 *et cetera* require representation components to be inherited, whereas [3] does not. However, we regard the representations in question in SQL:1999 *et cetera* as *actual* representations, rather than mere possible ones. Actual representation is an implementation issue, not a model issue. The implementer of our CIRCLE type is free to use an actual representation having A and B components instead of the R component of our possrep, provided, of course, that the prescribed consequences of the possrep are honored in the implementation.

As for specialization by constraint, which we spurned in Model 1 for fear of infeasibility, we now think this is no real problem. To implement it, we must be able to compute the most specific type of the result of evaluating an invocation of a scalar selector, by examination of type constraints. Suppose, for example, that type CIRCLE is defined thus (and here I am giving an airing to syntax we are currently considering for illustrative examples in our definition of Model 2):

```
TYPE CIRCLE ISA { ELLIPSE CONSTRAINT
   ( THE_A ( ELLIPSE ) =
        THE_B ( ELLIPSE ) ) }
    POSSREP { R LENGTH, C POINT }
```

(We use the term ISA to appeal to the notion that every value in the type being defined *is a* value in the specified supertype. The ISA specification is enclosed in braces to allow for multiple inheritance.)

Now consider again statement (14):

```
(14)  E := ELLIPSE ( LENGTH ( 5 ),
                     LENGTH ( 5 ),
                     XY_POINT ( 0, 0 )
                     ) ;
```

Like every invocation of a read-only operator, the expression on the right-hand side of the assignment operator in (14) has exactly one declared type, in this case ELLIPSE. The most specific type of its result must be some subtype of ELLIPSE, possibly ELLIPSE itself.

Our proposed method for determining the most specific type of a value *v* yielded by evaluation of an expression *e* is as follows. Consider the immediate subtypes of the declared type *dt* of *e*, taken in some arbitrary order. Test *v* for each of these subtypes in turn, to see if it satisfies the type constraint for that subtype. If *v* satisfies none of these constraints, then its most specific type is *dt*; otherwise, stop as soon as such a subtype *st* is found and repeat the process for the immediate subtypes of *st*.

This algorithm does rely on an assumption that the defining constraints are consistent with each other and with our model. For example, if *v* satisfies the constraint for some leaf type *lt*, then there is no other leaf type whose constraint is also satisfied by *v*. Also, if *v* satisfies the constraints of two distinct types, *t1* and *t2*, then its most specific type must be some subtype of the *least specific common subtype* of *t1* and *t2*. Let *lscs* be that common subtype. Then the constraint that defines *lscs* in terms of *t1* must be logically equivalent to the one defining it in terms of *t2*. We are currently considering the possibility of allowing these constraints to be implied. For example:

```
TYPE SQUARE ISA { RECTANGLE, RHOMBUS }
     POSSREP …
```

might be sufficient—the constraints IS_RHOMBUS (RECTANGLE) and IS_RECTANGLE (RHOMBUS), where IS_RHOMBUS and IS_RECTANGLE are truth-valued operators implicitly defined for parallelograms in general, are implied by the pairing of RHOMBUS and RECTANGLE in the ISA specification. The meanings of IS_RHOMBUS and IS_RECTANGLE are given by the type constraints that define RHOMBUS and RECTANGLE, respectively, in terms of PARALLELOGRAM. The reader can perhaps easily confirm that this idea generalizes to the case where three or more immediate supertypes are specified.

11. Why Model 2 is Not Feasible in SQL:1999 *et cetera*

The reason is strongly indicated by the subtitle of this article, "Object Identifiers and Inheritance Don't Mix!". Consider again statements (15) and (16), in the guise in which they appear in the following Java™ fragment:

```
        Ellipse e ;
        Circle c ;
(17)    c = new Circle ( 5 ,
                new Point ( 0, 0 ) ) ;
(18)    e = c ;
(19)    e.a = 6 ;
```

There is a subtle difference between on the one hand the pair of statements (15) and (16) and on the other the pair of statements (18) and (19). This difference is crucially important, as I will now explain.

Consider (17), ostensibly an assignment of a value of type `Circle` to a variable of that type. In fact, it is no such thing, and `c` is not a variable of type `Circle`! In Java™, `Ellipse` and `Circle` would be what are called *reference types*. This means that (17) in fact assigns to `c` not a circle, but instead the *object identifier* (oid) of a certain *circle object*. The `Circle` object in question comes into existence as a side effect of the given invocation of `new`, which returns the oid of that object.

Now consider (18). The effect of this is to assign to `e` the oid that is the current value of `c`. As a result, we note that `e` and `c` are both *referencing*, or, in jargon of long ago, *pointing at* the same object. Also in jargon of long ago, we would say that the object in question is a *shared variable* (and so, for that matter, is the object created by the invocation of `new Point` given as the second argument to the invocation of `new Circle`).

It's crunch time at last! The specified effect of (19) is to assign the length 6 to the `a` component of the object pointed at by the oid that is the current value of `e`. If Java™ were to embrace specialization by constraint, then the most specific type of that object would now have to be recomputed, and the system would discover that type to be `Ellipse`. As a consequence, the "circle variable" `c` now points to an object that is not a circle. This would have to be a run-time type error, and a type error of the very worst kind—the kind against which an application has no self-defense. Note that it is not possible to predict the occurrence of such an error merely by examination of the text of the statement whose invocation causes it. Note also that in an object-oriented database, that variable, `c`, might be a persistent variable anywhere in some large and widely distributed database. Unable to defend themselves against such situations, database applications could no longer meet the commonly required standards of *robustness*, given the possibility that such errors might arise.

And yet in Model 2 we propose to embrace specialization by constraint.

Our reason is simple. There is no concept of object identifier in our model. We have no pointers, no shared variables. We did not omit these things in order to be able to embrace an agreeable model of type inheritance. We omitted them simply because we continue to hold to the wisdom expressed in [2]. The reason Codd gave for spurning pointers in his Relational Model of Data was just that pointers are difficult to understand, confusing. He gave that reason at a time when type inheritance was not even being thought about (in a database context, at least). If Codd was right then, can we not claim to be even more right now?

12. SUMMARY AND CONCLUSION

I have described our motivation for formulating and proposing a rigorous type theory, incorporating the concept of type inheritance. I have described in outline the process by which we arrived at our first attempt, Model 1, published in [3] but later deprecated. Specifically, I related our close study of [8]'s "3 out of 4 rule" and our decision to follow others in favor of rejecting specialization by constraint in favor of keeping value substitutability, static type checking and mutability.

I have described Model 1 in outline and I have given our reasons for wanting to improve it. I have given notice of our intention to publish Model 2, which we claim successfully does embrace specialization by constraint, without, after all, sacrificing any of the other three desiderata. I have indicated the respects in which Model 2 will differ from Model 1 and with what favorable consequences.

We contend that, unlike Model 1 and the model(s) of SQL:1999, ODMG and Java™, Model 2 is an *agreeable* model, in that it conforms to normal human intuition about categorization of objects into types.

We have shown that Model 2 would be broken if the object identifier concept and its consequences were added to that model. We have recalled that object identifiers, being pointers of a kind, were shunned by Codd in his Relational Model of Data for reasons, not connected with type theory, that we still find to be sufficient reason for shunning them. We claim that because SQL:1999, ODMG and Java™ do support object identifiers, they cannot support an agreeable model of type inheritance, such as Model 2.

We conclude:

1. that the "3 out of 4" rule might more appropriately be replaced by a "4 out of 5" rule, with object identifiers as the fifth desideratum in the list from which any four but not all five can be chosen;
2. that *our* choice was effectively made for us in 1970;
3. that if a *truly* relational database language were to be devised and implemented, then that would be an opportunity for implementation of a well-defined and agreeable model of type inheritance.

ACKNOWLEDGMENTS

My thanks go to **Tom Pledger** of Peace Software, New Zealand, for his careful review and suggestions. Likewise to my coauthor, friend and fellow wanderer in Relationland, **Chris Date.**

REFERENCES

[1] R.G.G. Cattell and Douglas K. Barry (eds.): *The Object Database Standard: ODMG 2.0*. San Francisco, Calif.: Morgan Kauffman (1997).

[2] E.F. Codd: "A Relational Model of Data for Large Shared Data Banks", *CACM 13*, No. 6 (June 1970). Republished in "Milestones of Research", *CACM 26*, No. 1 (January 1982).

[3] C.J. Date and Hugh Darwen: "Foundation for Object/Relational Databases: The Third Manifesto". Reading, Mass.: Addison-Wesley (1998). (A 2nd edition is to appear in 2000).

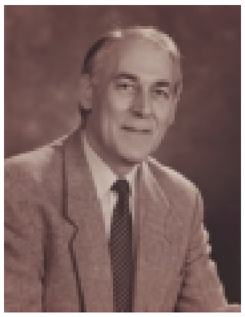
[4] James Gosling, Bill Joy, Guy Steele: "The Java™ Language Specification". Reading, Mass.: Addison-Wesley (1996).

[5] International Organization for Standardization (ISO): *Database Language SQL*. Document ISO/IEC 9075:1992.

[6] Jim Melton (ed.): "ISO Final Draft International Standard (FDIS) Database Language SQL — Part 2:Foundation (SQL/Foundation)" ISO/IEC FDIS 9075-2:1999.

[7] James Rumbaugh: "A Matter of Intent: How to Define Superclasses", *Journal of Object-Oriented Programming* (September 1996).

[8] Stanley B. Zdonik and David Maier: "Fundamentals of Object-Oriented Databases", in Stanley B. Zdonik and David Maier: *Readings in Object-Oriented Database Systems.* San Francisco, Calif.: Morgan Kauffman (1990).



Hugh Darwen is a database specialist working in Warwick, England for IBM United Kingdom Limited, for whom he has been involved in software development since 1967. He has been active in the relational database arena since 1978, from which date until 1982 he was one of the chief architects and developers of an IBM relational product called Business System 12—a DBMS that faithfully embraced the principles of the relational model. His writings include notable contributions to Date's *Relational Database Writings* series (Addison-Wesley, 1990, 1992), *A Guide to the SQL Standard* (4th edition, Addison-Wesley, 1997), and *Foundation for Object/Relational Databases: The Third Manifesto* (Addison-Wesley, 1998). He has been an active participant in the development of SQL international standards since 1988. One of his current special interests is in temporal databases. In his spare time he is a consultant to database course developers at the Open University, from whom he recently received the honorary degree of Master of the University, and he is a tutor to students on these courses. He is a visiting lecturer at several other British universities, one of whom, Wolverhampton University, in 1998, awarded him the honorary degree of Doctor of Technology. E-mail: Hugh_Darwen@uk.ibm.com.