And Now for Something Completely Computational

by

C. J. Date

July 17th, 2006

Myself when young did eagerly frequent
Doctor and Saint, and heard great Argument
About it and about; but evermore
Came out by the same Door as in I went.

—Edward Fitzgerald: *The Rubáiyát of Omar Khayyam*

ABSTRACT

The Third Manifesto, by Hugh Darwen and myself ("the *Manifesto*" for short), lays down a set of prescriptions and proscriptions regarding the design of a database programming language it calls **D** (see reference [9]). One prescription in particular—"OO Prescription 3"—reads as follows:

> **D** shall be **computationally complete**. That is, **D** may support, but shall not require, invocation from so-called host programs written in languages other than **D**. Similarly, **D** may support, but shall not require, the use of other languages for implementation of user-defined operators.

However, reference [1] argues that this prescription implies that the *Manifesto* is deeply flawed. To quote:[1]

> It's an error to make **Tutorial D** computationally complete because it creates a language with logical expressions that are provably not decidable—yet a decision procedure must exist for any logical expression to be evaluated.

Note: This quote refers to **Tutorial D,** not **D** as such, so I need to explain how **Tutorial D** relates to **D**. To begin with, the name **D** is generic—it's used in reference [9] to refer generically to any language that conforms to the principles laid down in *The Third Manifesto.* Thus, there could be any number of distinct languages all qualifying as a valid **D**. **Tutorial D** is intended to

[1] For reasons of clarity and flow I've edited most of the quotes in this paper, sometimes drastically so.

be one such; it's defined more or less formally in reference [9], and it's used throughout that book (and elsewhere) as a basis for examples. For definiteness I'll concentrate on **Tutorial D** myself (mostly) in the present paper, since that's what reference [1] does, but the discussions and arguments actually apply to any valid **D**.

DECIDABILITY

As noted in the abstract, reference [1] claims that **Tutorial D** suffers from a lack of decidability; more precisely, it claims that certain **Tutorial D** logical expressions aren't decidable. What does this mean?

Well, first, any expression can be thought of as a rule for computing a value; hence, a logical expression in particular can be thought of as a rule for computing a truth value. So a logical expression is an expression, formulated in some language *L,* that's supposed to denote either TRUE or FALSE.[2] Let *exp* be such an expression; to say *exp* is undecidable, then, is to say that although it's well formed—meaning it's constructed in full accordance with the syntax rules of *L* (it must be well formed, of course, for otherwise it wouldn't be an expression of the language)—there doesn't exist an algorithm that can determine in finite time whether *exp* evaluates to TRUE. By extension, the language *L* is said to be undecidable in turn if and only if there exists at least one *L* expression that's undecidable. I remark in passing that propositional calculus is decidable but predicate calculus is not.

If language *L* is decidable, then by definition there exists a general-purpose algorithm (a "decision procedure") for determining in finite time whether an arbitrary *L* expression evaluates to TRUE. By contrast, if *L* is undecidable, then no such algorithm exists. What's more, if *L* is undecidable, there isn't even an algorithm for determining ahead of time, as it were, whether a given *L* expression is decidable (if there were, the system could avoid the problem, at least in part, by not even attempting to evaluate expressions that are undecidable).

It follows from the foregoing that if **Tutorial D** in particular is undecidable, there'll be certain **Tutorial D** expressions, and hence certain **Tutorial D** queries (namely, ones including such expressions), that the system won't be able to deal with satisfactorily. On the face of it, therefore, the fact that **Tutorial D** is undecidable, if it is a fact, looks like a serious flaw.

[2] Naturally I limit my attention here to two-valued logic only.

COMPUTATIONAL COMPLETENESS

Reference [1] claims that it's specifically the fact that **Tutorial D** is computationally complete that makes it undecidable:

> If a given language incorporates predicate logic *and is computationally complete*, it's a logical consequence that some syntactically correct expressions of that language will be undecidable [*italics added*].

So what exactly does it mean for a language to be computationally complete?

Oddly enough, I was unable to find a definition of the term *computational completeness* in any of the fairly large number of computing references I examined. But many of them did include a definition of the term *computable function*, and I think it's a reasonable guess that a language is computationally complete if and only if it supports the computation of all computable functions. I'll take that as my working definition, anyway. So what's a computable function? Here are a couple of definitions from the literature:

- A computable function is a function that can be computed by a Turing machine in a finite number of steps [11].

- A computable function is a function that can be coded using WHILE loops [12].

COMPUTATIONAL COMPLETENESS IMPLIES UNDECIDABILITY

As I've said, reference [1] claims it's specifically the fact that **Tutorial D** is computationally complete that makes it undecidable. Here's an extended version of the extract I quoted near the beginning of the previous section:

> The problem is simply this: If a given language incorporates predicate logic and is computationally complete, it's a logical consequence that the set of syntactically correct expressions of that language will include self-referential expressions, and some of those will be undecidable. This is what Gödel showed in the process of constructing his first incompleteness theorem ... So, if **Tutorial D** is computationally complete, the set of syntactically correct **Tutorial D** expressions includes self-referential, undecidable expressions. I merely apply Gödel to **Tutorial D**.

Reference [1] appeals to Gödel's second theorem, too. Here's an extended version of the extract I quoted in the abstract to this paper:

> It's an error to make **Tutorial D** computationally complete because it creates a language with logical expressions that are provably not decidable—yet a decision procedure must exist for any logical expression to be evaluated (see Gödel's second incompleteness theorem). Is an example in **Tutorial D** more convincing than a proven theorem? Codd carefully avoided this trap, but *The Third Manifesto* does not!

I'll come back to the question of whether Codd "avoided this trap" in the next section. First, however, let me state Gödel's two theorems. *Note:* Actually those theorems can be stated in many different forms; the versions I give here are somewhat simplified—in fact, oversimplified—but they're good enough for present purposes. Note too that for the purposes of this discussion I'm regarding the terms *expression* and *statement* as interchangeable, even though they're not usually so regarded in conventional programming language contexts.

- *Gödel's First Incompleteness Theorem:* Let *S* be a consistent formal system that's at least as powerful as elementary arithmetic; then *S* is incomplete, in the sense that there exist statements in *S* that are true but can't be proved in *S*.

- *Gödel's Second Incompleteness Theorem:* Let *S* be a consistent formal system that's at least as powerful as elementary arithmetic; then the consistency of *S* can't be proved in *S*.

I confess I don't directly see the relevance of the second theorem to the arguments of reference [1], but the first clearly does support those arguments. What's more, since Gödel's proof of his first theorem (a) involves the explicit construction of a self-referential statement—in effect an arithmetic analog of the statement "This statement can't be proved in *S*"—and then (b) proves that precisely that statement can't be proved in *S* (and is therefore true!), it follows more specifically that the set of undecidable expressions in *S* includes certain self-referential expressions. So all right: If **Tutorial D** is computationally complete, it's undecidable, in the sense that its expressions include ones that are self-referential and undecidable.

DOES CODD "AVOID THE TRAP"?

So far we've seen that reference [1] claims, apparently correctly, that **Tutorial D,** being computationally complete, is undecidable. What's more, it also claims, at least implicitly, that Codd's relational algebra and relational calculus are decidable; in fact, they must be, since Codd's calculus is essentially an applied form of propositional calculus, which is decidable, and Codd's algebra is logically equivalent to his calculus.

Two asides here: First, relational calculus is usually thought of as an applied form of predicate calculus, not propositional calculus; however, the fact that we're dealing with finite systems means that it is indeed propositional calculus that we're talking about, at least from a logical point of view. Second, Codd's original calculus was actually less expressive than his algebra (i.e., there were certain algebraic expressions that had no equivalent in the calculus), but this deficiency in the calculus was subsequently remedied. The details need not concern us further in this paper.

As we've seen, reference [1] also claims that Codd "avoids the trap" of requiring computational completeness and thereby getting into the problem of undecidability. Now, that reference never actually comes out and states exactly how this goal—i.e., avoiding the trap—is to be achieved, but it does imply very strongly that it's a matter of drawing a sharp dividing line between the database and computational portions of the language:

> My desire is that **Tutorial D** should cleanly separate set theoretic semantics and computer language semantics[3] ... OO Prescription 3 should be restated to say that **D**'s application sublanguage shall be computationally complete and entirely procedural and **D**'s database sublanguage shall not be computationally complete, shall be entirely nonprocedural, and shall be decidable (though I would still be concerned about how these sublanguages interact with each other) ... [*Another attempt, later:*] Separate **D** into **RD** (the relational, nonprocedural part) and **CD** (the computational, procedural part). Then OO Prescription 3 should be restated to say that **RD** shall not be computationally complete and shall not invoke any operator that cannot be implemented in **RD** ... A relational database language **RD** must be decidable. It follows that it must not be computationally complete, nor should it invoke any procedure that cannot, in principle, be implemented in **RD**.

So do Codd's own language proposals abide by such restrictions? No! Consider the following quotes from Codd's own writings:

- *From the very first (1969) paper on the relational model* [2]: Let us denote the retrieval sublanguage by R and the host language by H ... R permits the specification for retrieval of any subset of data from the data bank ... The class of qualification expressions which can be used in a set specification is in a precisely specified ... correspondence with the class of well-formed formulas of the predicate calculus ... *Any arithmetic functions needed can be defined in H and invoked in R* [*italics added*].

- *From the revised version of the 1969 paper that appeared in 1970 in Communications of the ACM* [3]: Let us denote the data sublanguage by R and the host language by H ... R permits the specification for retrieval of any subset of data from the data bank ... [The] class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus ... *Arithmetic functions may be needed in the qualification or other parts of retrieval statements. Such functions can be defined in H and invoked in R* [*italics added*].

And Codd's paper on what he called Data Sublanguage ALPHA [4] had this to say:

> All computation of functions is defined in host language statements; all retrieval and storage operations in data sublanguage statements. A data sublanguage statement can, however, contain a use of a function defined in host statements ... An expandable library of functions which can be invoked in queries provides a means of extending the selective capability of DSL ALPHA.

The paper goes on to give several examples of the use of such functions, in both the "target list" and "qualification" portions of queries. So I would say that, in all three of these papers [2-4], Codd didn't just fail to "avoid the trap"—apparently, he didn't even think there was a trap to avoid in the first place.

Note: Commenting on an earlier draft of the present paper, the author of reference [1] claimed that Codd's separation of the languages *R* and *H* was sufficient to "avoid the trap." In

[3] On the issue of "set theoretic semantics and computer language semantics," see reference [8].

particular, he claimed that (a) one effect of that separation was that an invocation in *R* of a function defined in *H* could be regarded as a constant so far as *R* was concerned, and (b) the trap was avoided because functions defined in *H* had no access to the variables of *R*. On reflection, I don't understand these claims. In particular, the specific functions Codd uses in examples in reference [4] most certainly do have access to "the variables of *R*"—the functions in question include analogs of the familiar aggregate operators COUNT, SUM, AVG, MAX, and MIN, all of which are of course explicitly defined to operate on relations. (If the point is that these functions only read their operands and don't update them, then a precisely analogous remark could be made in respect of **Tutorial D,** so presumably that isn't the point.)

As an aside, I'd like to add that, for reasons it would be invidious to go into detail on here, I've never been much of a fan of the data sublanguage idea anyway (which is partly why the *Manifesto*'s OO Prescription 3 reads the way it does). As I wrote in reference [7]:

> Personally, I've never been entirely convinced that factoring out data access into a separate "sublanguage" was a good idea, [although it's] been with us, in the shape of embedded SQL, for a good while now. In this connection, incidentally, it's interesting to note that with the addition in 1996 of the PSM feature ("Persistent Stored Modules") to the SQL standard, SQL has now become a computationally complete language in its own right!—meaning that a host language as such is no longer logically necessary (with SQL, that is).

WHY WE WANT COMPUTATIONAL COMPLETENESS

Here again is OO Prescription 3 as originally stated:

> **D** shall be **computationally complete**. That is, **D** may support, but shall not require, invocation from so-called host programs written in languages other than **D**. Similarly, **D** may support, but shall not require, the use of other languages for implementation of user-defined operators.

Reference [1] commented on this prescription as follows:

> I don't understand the text beginning "That is"—it isn't a definition of what it would mean for a language to be computationally complete. Being invocable from, or being able to invoke, programs written in other languages does not make a language computationally complete.

I certainly agree that the text beginning "That is" isn't a definition of computational completeness; it wasn't meant to be, and some rewording might be desirable. Rather, it was meant to spell out certain consequences that follow if **D** is computationally complete—in other words, it was meant to explain why we thought computational completeness was a good idea. As Hugh wrote in his own portion of reference [1]:

> I hope the justification for our inclusion of computational completeness is clear. It is partly so that applications can be written in **D,** to avoid the problems inherent in writing

them in some other language, and partly to allow implementations of user-defined operators to be coded in **D**.

Later in the correspondence, in response to the criticisms I've already discussed, Hugh said this:

> We could perhaps have said something like this instead: **D** shall include comprehensive facilities for the implementation of database applications and user-defined operators. A computationally complete language would suffice for these purposes but **D** is not required to be computationally complete; nor are applications and user-defined operators required to be written in **D**.

But if we agree to back off from computational completeness, how far do we go?—i.e., where do we draw the line? How much computation can we safely support? If it's true that computational completeness just means being able to compute all computable functions, and a computable function just means a function that can be coded using WHILE loops, do we have to prohibit WHILE loops? If so, where does that leave us? *Note:* These questions are rhetorical, of course. My point, in case it isn't obvious, is that I don't think we *can* back off from computational completeness. What's more, Hugh agrees with me; his suggestion that **D** might not need to be computationally complete was never meant as more than a straw man.

But there's another issue I need to address under the rubric of why we wanted **D** to be computationally complete. Computational completeness implies among other things that relational expressions can include invocations of user-defined, read-only, relation-valued operators—operators whose implementation might be coded in **D** itself, perhaps using loops or other procedural constructs—and some critics seem to think that such a state of affairs is contrary to Codd's original intent that queries, etc., should all be expressed declaratively. However, we would argue that all read-only operator invocations are equally "declarative," regardless of where, how, by whom, and in what language(s) those operators are implemented (and regardless of whether they're relation-valued). By way of illustration, consider the following example:

```
OPERATOR TABLE_NUM ( K INTEGER )
   RETURNS RELATION { N INTEGER } ;
   ... implementation code ...
END OPERATOR ;
```

When invoked, this operator returns a relation representing the predicate "*N* is an integer in the range 1 to *K*" (the utility of such an operator is demonstrated in reference [6]). Surely, then, an invocation such as TABLE_NUM (3149) is equally "declarative" regardless of whether the implementation code (a) is written in **D** by the user doing the invoking, or (b) is written in **D** by some other user, or (c) is written by some user in some other language, or (d) is provided as part of the DBMS.

DOES IT ALL MATTER?

Reference [1] again:

> If **Tutorial D** is undecidable, sooner or later, whether by human user accident or by machine generation, an attempt will be made to evaluate an undecidable statement and the implementation will fail. You might object by pointing out that this does not happen in computationally complete languages such as Ada or Pascal or Fortran or Java. However, you would be quite wrong. It is actually quite easy to code an infinite procedural loop which no compiler can detect.

More specifically, as we've seen, reference [1] claims that a relational language must have an associated decision procedure ("a decision procedure must exist for any logical expression to be evaluated"). But is this claim correct? Predicate calculus has no decision procedure, but at least it's possible to come up with a procedure that's sound and complete. To paraphrase reference [10]:

> Given a well-formed formula of predicate calculus, such a procedure will correctly return TRUE if and only if that formula evaluates to TRUE; however, if the formula evaluates to FALSE, either the procedure will return FALSE or it will run forever. (In other words, if some formula is true, it's provably true; if it's false, however, it might not be provably false.)

In practice, therefore, we can incorporate such a procedure into the system implementation. Moreover, we can incorporate a time-out mechanism into that procedure, such that if evaluation of some given expression fails to halt after some predetermined period of time, the system can terminate evaluation and return a message to the user, along the lines of *Expression too complex*. (What it mustn't do, of course, is return either TRUE or FALSE! To do that would be to return what Codd—albeit writing in a very different context—called a "severely incorrect" result [5].)

To summarize, therefore:

- Clearly we would like a system in which all possible expressions can be evaluated in finite time.
- This objective can't be achieved if we insist on computational completeness.
- However, we are at least aware of this fact, and so we can plan for it.
- In particular, we can build code into the system that allows it to respond to certain queries by saying, in effect, "I can't answer this query because it's too complex."

In conclusion, I'd like to point out that:

- Inability to respond definitively to certain queries is a common occurrence in ordinary human discourse. We deal with such situations all the time. So having the system occasionally respond with an *Expression too complex* message doesn't necessarily mean the system is completely useless.

- In any case, even without computational completeness, it seems likely that there will exist queries that, though answerable in finite time in principle, might take so long to answer in practice that they are effectively unanswerable after all. In other words, the undecidability problem exists, in a sense, even without computational completeness. And our pragmatic fix for that problem (implementing a time-out mechanism) is therefore presumably needed anyway.

- Finally, if computational completeness leads to a lack of decidability, then it follows that conventional programming languages are undecidable. But we've lived with this problem for many years now, and I don't think it's led to any insuperable difficulties. Why should database languages be any different in this regard?

REFERENCES

1. Anon.: Private correspondence with Hugh Darwen (December 2005 - January 2006).

2. E. F. Codd: "Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks," IBM Research Report RJ599 (August 19th, 1969).

3. E. F. Codd: "A Relational Model of Data for Large Shared Data Banks," *CACM 13,* No. 6 (June 1970). Republished in *Milestones of Research—Selected Papers 1958-1982 (CACM 25th Anniversary Issue), CACM 26,* No. 1 (January 1983).

4. E. F. Codd: "A Data Base Sublanguage Founded on the Relational Calculus," Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego, Calif. (November 1971).

5. E. F. Codd and C. J. Date: "Much Ado About Nothing," in C. J. Date, *Relational Database Writings 1991-1994* (Addison-Wesley, 1995).

6. Hugh Darwen (writing as Andrew Warden): "A Constant Friend," in C. J. Date, *Relational Database Writings 1985-1989* (Addison-Wesley, 1990).

7. C. J. Date: *The Database Relational Model: A Retrospective Review and Analysis.* Reading, Mass.: Addison-Wesley (2000).

8. C. J. Date: "To Be Is to Be a Value of a Variable," *www.thethirdmanifesto.com* (July 2006).

9. C. J. Date and Hugh Darwen: *Databases, Types, and the Relational Model: The Third Manifesto* (3rd edition). Reading, Mass.: Addison-Wesley (2006).

10. Zohar Manna and Richard Waldinger: *The Logical Basis for Computer Programming, Volume 2: Deductive Systems.* Reading, Mass.: Addison-Wesley (1990).

11. Sybil P. Parker (ed.): *The McGraw-Hill Dictionary of Mathematics*. New York, N.Y.: McGraw-Hill (1994).

12. Eric W. Weisstein: *CRC Concise Encyclopedia of Mathematics.* Boca Raton, Fla.: Chapman & Hall / CRC (1999).

APPENDIX

In the body of this paper, I quoted reference [12] as defining a computable function to be one that can be coded using WHILE loops. Following this definition, that reference goes on to say:

> FOR loops (which have a fixed iteration limit) are a special case of WHILE loops, so computable functions can also be coded using a combination of FOR and WHILE loops. The Ackermann function is the simplest example of a function which is computable but not primitive recursive.

Let me elaborate briefly on these remarks. First, a function is said to be recursive if and only if it "can be obtained [*sic*] by a finite number of operations, computations, or algorithms" [11]. (Note that the term *recursive* here is not being used in the usual programming language sense; in fact, it seems to mean nothing more nor less than computable, as that term was previously defined.) Second, a function is said to be primitive recursive if and only if it can be coded using FOR loops only [12].

Now, I don't know in what sense the Ackermann function can be said to be "the simplest example" of a function that's recursive (or at any rate computable) but not primitive recursive. For interest, however, I give the definition of that function here (and I note that this definition in particular is certainly recursive in the usual programming language sense). Here it is: Let *x* and *y* denote nonnegative integers. Then the Ackermann function A(*x*,*y*) can be defined thus:

```
OPERATOR A ( X NONNEG_INT, Y NONNEG_INT ) RETURNS NONNEG_INT ;
     RETURN ( CASE
               WHEN X = 0 THEN Y + 1
                WHEN Y = 0 THEN A ( X - 1, 1 )
                ELSE A ( X - 1, A ( X, Y - 1 ) )
              END CASE ) ;
END OPERATOR ;
```

Warning: Please don't try to execute this algorithm on a real machine, not even for fairly small *x* and *y*.

***** End *** End *** End *****