To Be Is to Be a Value of a Variable

by

C. J. Date

with apologies to George Boolos
and his book Logic, Logic, and Logic
(I cribbed the title of this paper from an essay in that book)

If we want things to stay as they are,
things will have to change
—Giuseppe di Lampedusa

"Change" is scientific, "progress" is ethical;
change is indubitable,
whereas progress is a matter of controversy
—Bertrand Russell

July 17th, 2006

ABSTRACT

In reference [1], two writers, referred to herein as *Critics A* and *B,* criticize *The Third Manifesto* for its support for relation variables and relational assignment. This paper is a response to that criticism. Readers are expected to be familiar with the following concepts and terminology:

- A relation variable (relvar for short) is a variable whose permitted values are relation values (relations for short).

- Relational assignment is an operation by which some relation *r* is assigned to some relvar *R*.

Reference [14] explains these notions in detail, using a language called **Tutorial D** as a basis for examples.

References

WHY WE WANT RELVARS

As noted in the abstract, the term *relvar* is short for relation variable. It was coined by Hugh Darwen and myself in reference [9], the first published version of *The Third Manifesto;* Codd's first papers on the relational model [3-4] used the term *time-varying relation* instead, but "time-varying relations" are just relvars by another name. Of course, we don't claim to be the first to recognize this fact, but we do believe we were the first to draw wide attention to it. *Note:* Reference [2], which predated the first version of *The Third Manifesto* by several years, also clearly distinguished between relations and relvars (it called them tables and table variables, respectively). However, it did so only as a direct consequence of comments by myself on an earlier draft, which didn't.

We believe further that relvars and the related notion of relational assignment are essential if we're to be able to update the database. Note that variables and assignment go hand in hand (we can't have one without the other)—to be a variable is to be assignable to, to be assignable to is to be a variable. Note further that "assignable to" and "updatable" mean exactly the same thing; hence, to object to relvars is to object to relational updating (equivalently, to object to relational updating is to object to relvars).

Perhaps a little additional explanation is needed here. Most people, if they think about relational updating at all, probably think about the conventional INSERT, DELETE, and UPDATE operators, not about relational assignment—especially as SQL in particular doesn't support relational assignment, though it does support INSERT, DELETE, and UPDATE, of course. But INSERT, DELETE, and UPDATE are all in the final analysis just shorthand for certain relational assignments. For example, suppose we're given the usual suppliers-and-parts database (see Fig. 1 for a set of sample values).* Then the **Tutorial D** INSERT statement

──────────

* The discussion and examples that follow are taken from reference [10].

──────────

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

Fig. 1: The suppliers-and-parts database—sample values

```
    INSERT SP RELATION
         { TUPLE { S# S#('S3'), P# P#('P1'), QTY QTY(150) },
           TUPLE { S# S#('S5'), P# P#('P1'), QTY QTY(500) } } ;
```

is shorthand for the relational assignment

```
    SP := ( SP ) UNION ( RELATION
          { TUPLE { S# S#('S3'), P# P#('P1'), QTY QTY(150) },
            TUPLE { S# S#('S5'), P# P#('P1'), QTY QTY(500) } } ) ;
```

Likewise, the **Tutorial D** DELETE statement

```
    DELETE S WHERE CITY = 'Athens' ;
```

is shorthand for the relational assignment

```
    S := S WHERE NOT ( CITY = 'Athens' ) ;
```

And the **Tutorial D** UPDATE statement

```
    UPDATE P WHERE CITY = 'London'
         ( WEIGHT := 2 * WEIGHT, CITY := 'Oslo' ) ;
```

(a little trickier, this one) is shorthand for the relational assignment

```
P := WITH ( P WHERE CITY = 'London' ) AS T1,
          ( EXTEND T1
            ADD ( 2 * WEIGHT AS NW, 'Oslo' AS NC ) ) AS T2,
          ( T2 { ALL BUT WEIGHT, CITY } ) AS T3,
          ( T3 RENAME ( NW AS WEIGHT, NC AS CITY ) ) AS T4,
          ( P MINUS T1 ) AS T5 :
     T5 UNION T4 ;
```

It should be clear, therefore, that relational assignment is fundamentally the only relational updating operator we need. For that reason, I'll focus on relational assignment as such for the remainder of this paper. Also, throughout the paper from this point forward, I'll take (a) the unqualified term *assignment* to mean relational assignment specifically and (b) the unqualified term *relation* to mean a relation value specifically (except possibly in quotes from other writers).

***CRITIC A*'S OBJECTIONS**

Relvars and assignment are criticized in a lengthy series of messages from *Critics A* and *B* to Hugh Darwen [1]. The overall exchange was sparked off by a question on an issue only tangentially related to the matter at hand (and I'll therefore ignore the substance of that issue in what follows). In his reply to the questioner, *Critic A* said this:[1]

> I and [*Critic B*] do not subscribe to relvars and think Codd did not either.

Hugh responded:

> I'm baffled by your nonsubscription to relvars ... Don't you subscribe to INSERT, DELETE, UPDATE, and relational assignment? Codd certainly did. The target operand for all of these operations is a relation variable (relvar for short).

To which *Critic A* replied:

> More precisely, we don't subscribe to explicit relvars and Codd used "time-varying relations" to avoid them ... Bearing in mind that simplicity was one of Codd's main objectives, we think he may have refrained intentionally from introducing relvars. He was obviously aware of the time dimension of databases, yet as far as we have been able to determine, he never included time-variance semantics in his formal model. Had he done so, the language of sets and mathematical relations would have been rather strained because, as Date himself points out, every object in the language has a fixed value. Since relationships within and among Codd's relations are evaluated at a point in time, this permits the use of set semantics ... While conceptually Codd's "time-varying relation" has to be something like a relvar, the "gloss" permitted Codd to stick to simple sets (which

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

> cannot change), yet still contend with updates ... It is, perhaps, significant that later, in his RM/T paper, he referred to insert-update-delete as "transition rules," not operations.

And in a subsequent email he (*Critic A*) went on to say:

> Please note that it is not claimed there are no relvars involved. The only claim is that it is not a good idea to deal with them explicitly in the data language, because it creates complexity due to problems with unfixed sets. It's hard to believe that Codd did not think about variables, and that he used the term "time-varying relation" lightly.

——— ♦ ♦ ♦ ♦ ♦ ———

At this point I'd like to interject some blow-by-blow responses of my own to these various remarks of *Critic A*'s. I've repeated and numbered those remarks for purposes of reference.

1. More precisely, we don't subscribe to explicit relvars.

   This statement seems to suggest that *Critic A* does subscribe to "implicit" relvars, whatever they might be. So apparently relvars are bad only if they're explicit. I don't understand this position.

2. Codd used "time-varying relations" to avoid [explicit relvars].

   There are two ways to interpret this remark. The first is: Codd used the *concept* of "time-varying relations" in order to avoid having to deal with the *concept* of relvars (explicit or otherwise). If this interpretation is the intended one, then I'd like to know exactly what the difference is between these two concepts; our critics claim a difference exists, but they never seem to come out and say what it is.

   The second interpretation is: Codd used the *term* "time-varying relations" in order to avoid having to use the *term* "relvars" (again, explicit or otherwise). If this interpretation is the intended one, then I simply don't believe it. I worked with Codd for many years and knew him well, and I had many discussions with him on this very point. While I don't think I can do complete justice to his position on the matter, I can at least state with some authority that there was no hidden agenda behind his use of the term "time-varying relation"; it was just the term he used, that's all, and I don't think he attached any great significance to it.

   More particularly (and contrary to both of the foregoing possible interpretations of *Critic A*'s remark), the papers (references [3] and [4]) in which Codd first used the term contain not the slightest hint that he introduced it to avoid discussing variables and/or updating. *Au contraire*, in fact: In both of those papers, he explicitly discussed the question of relational updating. To quote: "Insertions take the form of adding new elements to declared relations ... Deletions ... take the form of removing elements from declared relations." What's more (in case you might be wondering what Codd meant by the term *declared relation*), references [3] and [4] both make it clear that a declared relation is a named relation ("time-varying," of course) that (a) is explicitly declared as such to the system, (b) is described in the system catalog, (c) can be updated (so the declared name denotes different relations—that is, different relation *values*—at different times), and (d) can be referenced in queries (and constraints, presumably). That looks like a relvar to me, and an explicit one to boot.

3. Bearing in mind that simplicity was one of Codd's main objectives, we think he may have refrained intentionally from introducing relvars.

   I find no evidence in any of his writings that Codd ever intended any such thing; in fact, I find a great deal of evidence to the contrary—not only in the remarks just quoted regarding insertions and deletions and declared relations, but in numerous remarks elsewhere as well.

4. He was obviously aware of the time dimension of databases, yet as far as we have been able to determine, he never included time-variance semantics in his formal model.

   If "time-variance semantics" merely means that Codd's time-varying relations vary over time, there is clear evidence—not solely in the name—that he did include such semantics. In particular, he certainly included relational assignment "in his formal model," a point I'll come back to later.

5. Had he done so, the language of sets and mathematical relations would have been rather strained because, as Date himself points out, every object in the language has a fixed value.

   I don't know what this means, nor do I know what writings of my own are being referred to here.

6. Since relationships within and among Codd's relations are evaluated at a point in time, this permits the use of set semantics.

   The phrases "set semantics" and a slight variant, "set theoretic semantics," appear repeatedly in reference [1], but I have little idea as to what they mean. From other remarks in reference [1] I can guess they refer to something that includes set operators such as union and intersection but excludes assignment; but then why not talk about (e.g.) "arithmetic semantics," meaning something that includes arithmetic operators such as "+" and "*" but excludes assignment? (I won't repeat these questions every time one of the unclear phrases appears, letting this one paragraph do duty for all.) Overall, I don't think this remark of *Critic A*'s means anything other than that the value of a relvar at any given point in time is a relation (whose body is a set, of course: namely, a set of tuples). If that is indeed what it means, then of course I agree, but I can't attach any special significance to it.

7. While conceptually Codd's "time-varying relation" has to be something like a relvar, the "gloss" permitted Codd to stick to simple sets (which cannot change), yet still contend with updates.

   I agree with Hugh's response on this one. To quote:

> Well, somebody will have to explain to me what the difference is [*between a relvar and a "time-varying relation"*] ... If it walks like a duck, swims like a duck, flies like a duck, and quacks like a duck, what is it?

See also my own earlier comments on this same issue. *Note:* I might add that I don't really understand what's meant by the term "gloss" here, either, but perhaps it's not important.

8. It is, perhaps, significant that later, in his RM/T paper, he referred to insert-update-delete as "transition rules," not operations.

   No, he didn't. What he actually said was this [6]:

   All insertions into, updates of, and deletions from ... relations are constrained by the following two rules [*and he goes on to give definitions of the entity and referential integrity rules. Then he explicitly states that the relational model includes those two rules, and he refers to them generically as*] the insert-update-delete rules.

   Note the explicit reference to "insertions into, updates of, and deletions from" relations! (Incidentally, the paper continues to refer to the target of such operations as "time-varying relations.")

9. Please note that it is not claimed there are no relvars involved. The only claim is that it is not a good idea to deal with them explicitly in the data language, because it creates complexity due to problems with unfixed sets.

   To the extent that I understand these remarks (which isn't very far), they just look like arm waving to me. See my response to *Critic A*'s remark no. 1.

10. It's hard to believe that Codd did not think about variables, and that he used the term "time-varying relation" lightly.

    No, it's not. See my response to *Critic A*'s remarks nos. 2 and 4.

——— ♦ ♦ ♦ ♦ ♦ ———

I've already quoted part of Hugh's response to *Critic A*'s remarks. That response continues:

> I thought "relational assignment" was Codd's term, and one of his twelve rules ... Codd's accounts of assignment, insert, update, and delete on pp 87-94 of the RM/V2 book look indistinguishable from those of **Tutorial D** ...

Well, I can confirm that Codd used the term *relational assignment* in "the RM/V2 book" [8], though not in fact in "the twelve rules" paper [7]. (One of those rules does have to do with INSERT, DELETE, and UPDATE, but there's no rule regarding assignment as such.) But he certainly included the *concept* of assignment, and explicit syntax for that concept, much earlier than that—in the RM/T paper [6], to be specific (which appeared in 1979), and possibly earlier still.

As an aside, I have to say that it's not at all obvious that RM/V2's facilities in this area are "indistinguishable from those of **Tutorial D**" (and indeed they aren't; for example, the RM/V2 facilities do include the idea that certain deletes can cause the introduction of nulls into the database, while at the same time they don't include support for multiple assignment). What's more, the text on pages 87-94 of the RM/V2 book contains much material not directly related to the semantics of the operators as such, including many details that don't belong in an abstract model at all—e.g., "Whenever rows are withheld by the DBMS from insertion (to avoid duplicate rows in the result), the *duplicate row indicator* is turned on"; "If one or more indexes exist for the target relation, the DBMS will automatically update these indexes to support the inserted rows"; and so on. It also contains several prescriptions that are in direct conflict with *The Third Manifesto*—e.g., "The domain of any column of T in which the values are derived by means of a function is identified [in the catalog] as *function-derived,* because the DBMS usually cannot be more specific than that"; "The relational model includes the *cascading option* in some of its manipulative operators";[2] and so on. All of that being said, however, I do of course agree with Hugh that the general functionality being defined in this part of the RM/V2 book is essentially similar to that found in the analogous portions of **Tutorial D**.

To all of the above I add that as early as 1971 Codd was proposing explicit support for INSERT, DELETE, and UPDATE (albeit not for assignment as such); I refer to his paper on "Data Sublanguage ALPHA" [5], in which 12 examples (out of a total of 32, or nearly 40 percent) were updating examples specifically.

CRITIC B'S OBJECTIONS

After the exchanges between *Critic A* and Hugh discussed above, *Critic B* joined the correspondence (effectively taking over from *Critic A*, who didn't contribute any further). In his first message, *Critic B* said this among other things:

> The conflation of set theoretic language (which has only equivalence) and a computational language (which has both assignment and equivalence) results in muddy semantics, which neither Hugh nor Chris have discussed or even acknowledged. Furthermore, neither seem to have applied any of the vast literature on nondecidability and incompleteness to *The Third Manifesto*.

Well, it's true that *The Third Manifesto* prescribes, and **Tutorial D** (like every other imperative language I know) supports, "both assignment and equivalence." In fact, the *Manifesto* prescribes, and **Tutorial D** supports, all three of the following:

- *Logical equivalence:* If *p* and *q* are predicates, the equivalence (*p*) EQUIV (*q*)—not meant to be actual **Tutorial D** syntax—is a predicate also, evaluating to TRUE if and only if *p* and *q* both evaluate to the same truth value.

- *Value equality "=":* Values *v1* and *v2* are equal if and only if they're the very same value.

[2] *The Third Manifesto* does not prohibit "cascading options" that are specified declaratively, but Codd is suggesting here that they might be specified procedurally instead.

- *Assignment ":=" (relational or otherwise):* The assignment *V* := *v* causes the specified value *v* to be assigned to the specified variable *V* (after which, the comparison *V* = *v* is required to evaluate to TRUE).

I'd like to elaborate on value equality in particular, since certain subsequent remarks of *Critic B*'s suggest there might be some breakdown in communication in this area. As I've said, values *v1* and *v2* are equal if and only if they're the very same value (and I note in passing that the term *identity* might reasonably be used instead of equality for this concept). It's our position, reflected in the *Manifesto,* that any given value—e.g., the integer 3—exists (a) for all time and (b) exactly once in the universe (as it were), but that many distinct *occurrences* or *appearances* of that given value can exist simultaneously, in many different places. And if two such "places" happen to contain appearances of the same value at the same time, then comparing those two "places" for equality will give TRUE (they'll "compare equal") at that time.[3] Here's some text from reference [10] that explains the overall situation:

<quote>

Observe that there's a logical difference between a value as such and an appearance of that value—for example, an appearance as the current value of some variable or as some attribute value within the current value of some relvar. Each such appearance consists internally of some physical representation of the value in question (and distinct appearances of the same value might have distinct physical representations). Thus, there's also a logical difference between an appearance of a value, on the one hand, and the physical representation of that appearance, on the other; there might even be a logical difference between the physical representations used for distinct appearances of the same value. All of that being said, however, it's usual to abbreviate *physical representation of an appearance of a value* to just *appearance of a value*, or (more often) just *value*, so long as there's no risk of ambiguity in doing so. Note that *appearance of a value* is a model concept, whereas *physical representation of an appearance* is an implementation concept—for example, users certainly might need to know whether two variables contain appearances of the same value, but they don't need to know whether those appearances use the same physical representation.

Example: Let N1 and N2 be variables of type INTEGER. After the following assignments, then, N1 and N2 both contain an appearance of the integer value 3. The corresponding physical representations might or might not be the same (for example, N1 might use a binary representation and N2 a packed decimal representation), but it's of no concern to the user either way.

```
N1 := 3 ;
N2 := 3 ;
```

</quote>

[3] So we might say we have here an example of yet another kind of equality, which we might call *appearance equality*. No such term is used in the *Manifesto,* however.

What if anything is wrong with the foregoing state of affairs? *Note:* If (as *Critic B*'s next sentence might suggest) the answer to this question is that it gives rise to undecidability, then I've already dealt with that issue in a couple of companion papers [11-12], and I won't discuss it further here. But I can't tell from the quoted extract whether the problem that *Critic B* is referring to is indeed that one.

I'd also like to know exactly what's "muddy" about the semantics of **Tutorial D**. Hugh asked the same question:

> Please justify by showing concrete examples in **Tutorial D** where our "semantics" are "muddy." Please also explain what you think it takes for semantics to be muddy. I understand indeterminacy (as found in SQL), but I believe we have none of that.

Critic B never explicitly responded to these requests, as far as I can tell, unless the following is a response:

> Your request that I explain what **Tutorial D** does wrong through examples in **Tutorial D** is absurd! You cannot give examples in any language of what that language does NOT do!

I'll come back to these remarks of *Critic B*'s in a little while.

Anyway, Hugh wrote a long response to *Critic B*'s complaint, of which the following is the substance:

<quote>

> If the database language has no named relvars, how are updates expressed in it, and how are constraints expressed? And how are queries expressed? ... The answers must be accompanied by examples in some concrete syntax. This requirement is a stringent one and I might not respond to a response that does not attempt to address it. The syntax should be based, where appropriate, on relational algebra ...
>
> We have assignment so that the database can be updated. As far as the database is concerned, assignment is restricted to relational assignment only, because relation variables are the only kind allowed in the database ... A proposal to do away with relation variables needs to demonstrate two very important things: first and foremost, an alternative way of updating the database; second, the advantages of this alternative way over assignment to relvars.

</quote>

Critic B returned to the fray:

<quote>

> To clarify, I have NOT proposed doing away with the concept of relation variables per se ...

Your question goes to the heart of the very great difference in semantics between set theoretic and computational languages ... The set theoretic analog of "updating" semantics is two sets (e.g., {A} and {B}) connected by a "set transformation" or "transition" rule ... Semantically, this is VERY different from saying that {A} becomes {B} via some update operator because—in set theoretic language—{B} does not replace {A} and so there is no assignment of values to some variable. Instead both always exist but are merely related in a known way.

The problem created by combining set theoretic language and computational language semantics in some completely unspecified manner makes *The Third Manifesto* as flawed as the NULL problems in SQL!

Your request that I explain what **Tutorial D** does wrong through examples in **Tutorial D** is absurd! You cannot give examples in any language of what that language does NOT do!

In **Tutorial D,** I do not know how to interpret "equivalence"—sometimes you seem to want the set theoretic concept (i.e., an assertion of identity) and sometimes you seem to want the computational concept (an assertion of value equivalence). If the first is not intended, then how does **Tutorial D** support inference? And if it is, how do you square this with assignment, which is obviously at odds with the set theoretic semantics for which there is no concept of variable?

</quote>

I have some blow-by-blow responses of my own to all this:

1. To clarify, I have NOT proposed doing away with the concept of relation variables per se.

   This claim seems to be related to *Critic A*'s remark to the effect that (apparently) explicit relvars are bad but implicit ones might be OK. I still fail to understand what exactly is being proposed here.

2. Your question goes to the heart of the very great difference in semantics between set theoretic and computational languages.

   The question referred to is, I presume, the one in which Hugh asks how updates are to be done without relvars; if not, then I don't understand.

3. The set theoretic analog of "updating" semantics is two sets (e.g., {A} and {B}) connected by a "set transformation" or "transition" rule.

   I note that *Critic A* also referred to transition rules (though his reference was incorrect).

4. Semantically, this is VERY different from saying that {A} becomes {B} via some update operator because—in set theoretic language—{B} does not replace {A} and so there is no assignment of values to some variable. Instead both always exist but are merely related in a known way.

First, *The Third Manifesto* never talks in terms of "one set becoming another"; rather, it talks in terms of a variable which has one value at one time and another at another. Second, it also never talks in terms of one set replacing another; since all values "always exist," all sets also "always exist," a fortiori (in fact, sets *are* values). However, it does talk in terms of a variable being updated, which means the *appearance* of one value (in that variable) is replaced by an *appearance* of another. In fact, it tries very hard to be precise over such matters—over the logical difference, in particular, between a value as such and an appearance of such a value in some context, as I tried to explain some pages back—and it's truly frustrating to be so roundly misunderstood. Overall, these two sentences of *Critic B*'s just look like an attempt to state, fuzzily, what the *Manifesto* states very precisely.

5. The problem created by combining set theoretic language and computational language semantics in some completely unspecified manner makes *The Third Manifesto* as flawed as the NULL problems in SQL!

What exactly is it in *The Third Manifesto* that's "completely unspecified"? If anything's unspecified here, I'd have to say it's the meaning of "computational language semantics"—not to mention "set theoretic semantics," a notion I've already commented on. Also, what exactly does "as flawed as the NULL problems in SQL" mean? Nulls give rise to a many-valued logic, which most authorities agree causes horrible problems; but I'm not aware that the *Manifesto*'s insistence on "computational language semantics" necessitates any departure from conventional two-valued logic. At best, therefore, the reference to nulls is a red herring, and the claim that the *Manifesto* is "as flawed as the NULL problems in SQL" is an apples and oranges comparison.

Note added later: It occurs to me that the phrase "the problem created by combining set theoretic language and computational language semantics" might refer to something we categorically prohibit: namely, the possibility that a new value might be assigned to some variable during the process of evaluating some expression that involves that very same variable. We agree that allowing such a possibility could have adverse consequences (though some languages do in fact permit it). For that reason, any language that's supposed to conform to *The Third Manifesto* is required to satisfy the following prescriptions among others (and of course **Tutorial D** does satisfy these prescriptions):

- Syntactically, no assignment is an expression; more generally, no update operator invocation is an expression.

- Syntactically, therefore, no expression (no relational expression in particular) is allowed to include either an assignment or, more generally, an update operator invocation of any kind.

- By contrast, an expression (a relational expression in particular) is allowed to include a read-only operator invocation. However, such an invocation is itself fundamentally just shorthand for another expression; by definition, therefore, it includes no assignments and no update operator invocations of any kind.[4]

It follows from all of the above that if a given relational expression *exp* includes any references to some relvar *R*, then throughout evaluation of *exp* those references all denote the same thing: namely, the relation *r* that's the value of *R* immediately before evaluation of *exp* begins.

6. Your request that I explain what **Tutorial D** does wrong through examples in **Tutorial D** is absurd! You cannot give examples in any language of what that language does NOT do!

Well, I thought the point was (see references [11-12]) that **Tutorial D** allows expressions that can't be evaluated. If so, it must be possible to give an example of such an expression. Now, I agree it might be difficult to do so—I mean, the expression might be extremely complex—but *Critic B* is saying it's impossible. So perhaps *Critic B* is referring to something else that **Tutorial D** "does wrong." In fact, I think he must be—since he goes on to suggest that there's something the language "does NOT do," and allowing expressions that can't be evaluated is something it does do (at least according to *Critic B*).

When the foregoing points are clarified, I'd then like to know why analogous criticisms don't apply to the hypothetical language described in Codd's original papers [3-4] or to his ALPHA language [5]. And assuming I'm right in thinking those criticisms do apply, I'd also like to see a language to which they don't.

7. In **Tutorial D,** I don't know how to interpret "equivalence"; sometimes you seem to want the set theoretic concept (e.g., an assertion of identity) and sometimes you seem to want the computational concept (an assertion of value equivalence). If the first is not intended, then how does **Tutorial D** support inference? And if it is, how do you square this with assignment, which is obviously at odds with the set theoretic semantics for which there is no concept of variable?

I'm afraid I'm far from fully understanding these remarks. I *think* what *Critic B* here calls "assertion of identity" is what we call equality. I *think* what *Critic B* here calls "assertion of value equivalence" is what I earlier suggested (in a footnote) might be called "appearance equality." I've already tried to explain these constructs (viz., equality and "appearance equality"), and I believe the *Manifesto* is perfectly explicit on when and where they can be used and what their semantics are. As for "[the *Manifesto*] supporting inference": I *think* what *Critic B* is referring to here is the process of determining the value of a relational expression (in particular, the process of responding to a query). If so, then I believe the *Manifesto* is perfectly explicit on what's involved in that process.

[4] The code that implements a given read-only operator is always logically equivalent to a single RETURN statement, the operand to which is itself formulated as an expression. (While that implementation code might in fact be written in such a way as to update certain variables that are purely local to the operator in question, such updates have no lasting effect.) Thus, such an operator cannot and does not update anything in its environment; in particular, it cannot and does not update anything in the database.

> What's more, I fail to see how assignment and "the concept of variable" come into the picture, since—as I tried to explain a little while back—neither has any role to play in that process.

———— ♦♦♦♦♦ ————

In a subsequent message, *Critic B* said this:

> My desire is not to introduce a database language with no variable names, etc., but that **Tutorial D** should cleanly separate set theoretic semantics and computer language semantics. You want a single language which has both, but I don't believe this is possible unless (for example and at least) truth value equivalence is distinct from cardinal and ordinal value equivalence.

As I said earlier, **Tutorial D** has logical equivalence (which is presumably the same as what *Critic B* here calls truth value equivalence), together with value equality,[5] together with assignment (which *Critic B* previously at least suggested was also a kind of equivalence). Now he additionally talks about "cardinal and ordinal value equivalence." I have no idea whether or not this is one of the three kinds **Tutorial D** has; I don't know whether "cardinal and ordinal value equivalence" is one kind or two; and I don't even know whether *Critic B* thinks it would be good or bad if **Tutorial D** supported it (or them). Anyway, Hugh responded:

> I have explained what we mean by "equals," in response to certain statements from you that indicated you were worried that we had two different kinds. (I didn't understand both of the two kinds, but our only kind appears to be the one you want. See RM Prescription 8.)

What Hugh here calls "our only kind" is specifically value equality, the semantics of which are precisely specified in *The Third Manifesto*'s RM Prescription 8. *Critic B* replied:

> I realize you don't understand that there are two (actually many) kinds of "equal" ... As best I can guess, your ability to think in purely set theoretic terms when talking about **Tutorial D** is mentally blocked. Let me simply say that value equivalence is not the same as identity. Value refers to a comparison of measures of a quantitative property, while identity pertains to what mathematicians often call entities ("things").

Well, I'm going to have to repeat some things I've already said (and I apologize up front for the repetitiousness) ... but I strongly suspect from these remarks that *Critic B* hasn't taken on board exactly what *The Third Manifesto* means by the term *value*. I also suspect that what he calls "value equivalence" is what we mean when we talk of equality of distinct *appearances* of the *same* value (where we would say that—by definition—there's just one value, as such). I further suspect that this misunderstanding on his part (of our use of terms) has led him into a criticism that has no basis in fact. I also think, contrary to what *Critic B* is saying here, that our "value equivalence" (I used the term "value equality" earlier) *is* "the same as identity": Two appearances are equal ("value equal"?) if and only if they're appearances of the *identical*

[5] And possibly "appearance equality," too.

value. As for the notion that there are many kinds of equality: Well, it might be true (I really don't know) that many kinds can be defined, but I think the important one is the one we define in RM Prescription 8—and that's the one we appeal to, explicitly or implicitly, whenever we talk about equality as such in the context of *The Third Manifesto*.

In the same message, *Critic B* also says this:

I have not stated how I think updates to the database should be expressed, except that we can safely use the set theoretic representation as having both a "canonical" method and a "canonical" semantics. I object to assignment because I see it as being at odds with the set theoretic representation and importing a "before and after semantics" which is inherently procedural.

My responses:

1. I have not stated how I think updates to the database should be expressed.

   Well, as I said earlier (quoting Hugh), a proposal to do away with relation variables needs to demonstrate two very important things: first and foremost, an alternative way of updating the database; second, the advantages of this alternative way over assignment to relvars. It's truly frustrating to be told over and over that our approach doesn't work—especially without being told clearly why it doesn't work, and especially when it's essentially the same as the approach supported by all imperative languages since programming languages were first invented—without at the same time being told about some alternative approach that does work.

2. We can safely use the set theoretic representation as having both a "canonical" method and a "canonical" semantics.

   The significance of these observations is unclear to me.

3. I object to assignment because I see it as being at odds with the set theoretic representation and importing a "before and after semantics" which is inherently procedural.

   Assume for the sake of the discussion that (a) it's true that set theory has no notion of assignment and that (b) it's true that updates are a requirement. (For my part I have no difficulty in accepting either of these assumptions.) Then the obvious conclusion is not that assignment is inherently flawed; rather, it's that set theory by itself is inadequate as a theoretical basis for a database programming language. However, *Critic B* asserts that assignment and set theory (or "the set theoretic representation") are actually at odds with each other—i.e., they're actually in conflict, suggesting that if we support one we can't support the other. If this is true, then so much the worse for set theory; but frankly, I don't see why it's true. *Note:* Replace "set theory" by "logic" throughout the foregoing remarks, and the resulting argument is something I would also sign on to.

   What's more, the notion of "before and after semantics" is indeed implied by assignment. More significantly, however, it's implied by—*derives from* might be a better way of putting it—the fundamental way time works in our universe! (I suppose we might say that being "inherently procedural" derives from the way time works in our universe, too, if we could agree that "procedural" just means performing one action after another, in sequence; but the problem here is that the label "procedural" is usually taken to mean "*low-level* procedural" and hence is used, almost always, in a pejorative sense.) If set theory can't deal with "before and after semantics," then so much the worse for set theory. *Note:* Again, replace "set theory" by "logic" throughout the foregoing remarks and the resulting argument is something I would also sign on to.

MULTIPLE ASSIGNMENT

The Third Manifesto prescribes not just assignment per se but what it calls *multiple* assignment. Multiple assignment is an operation that allows several individual assignments all to be performed "simultaneously," as it were, without any integrity checking being done until all of those individual assignments have been executed in their entirety. For example, the following "double DELETE" is, logically, a multiple assignment operation:

```
DELETE S  WHERE S# = S#('S1') ,
DELETE SP WHERE S# = S#('S1') ;
```

Note the comma separator after the first DELETE, which indicates syntactically that the end of the overall statement has not yet been reached.

In reference [1], *Critic B* raises several questions about multiple assignment. To quote:

I am uncertain as to how you intend multiple assignment to be implemented. If there are, e.g., five individual assignments, are they processed in order as stated from top to bottom or is the order arbitrary or are they expected to be processed in parallel? Your rewrite algorithm for eliminating multiple references to the same variable raises more issues than it solves. At best, it seems to assume there are no side effects among the individual assignments, so that order does not matter. If this is the assumption, then clearly there are certain ordered sets of assignments (normally coded as transactions) that cannot be rewritten as a multiple assignment because they will produce a result different than that which was originally intended ... I like the idea of multiple assignment but not at the expense of transactions and therefore not at the expense of deferred constraint checking.

Some blow-by-blow responses:

1. I am uncertain as to how you intend multiple assignment to be implemented.

   We expect it to be implemented as specified. The semantics are specified in *The Third Manifesto* [14] and also in a standalone paper [13].

2. If there are, e.g., five individual assignments, are they processed in order as stated from top to bottom or is the order arbitrary or are they expected to be processed in parallel?

This question is fully answered in references [13] and [14]. For the record (and simplifying slightly), the basic idea is that (a) the expressions on the right-hand sides of the individual assignments are evaluated (in arbitrary order, because the order makes no difference) and then (b) the individual assignments to the variables on the left-hand sides are executed in sequence as written.

3. Your rewrite algorithm for eliminating multiple references to the same variable raises more issues than it solves.

   References [13] and [14] do include a "rewrite algorithm" for combining—not eliminating!—"multiple references to the same variable." If that algorithm truly does raise "more issues than it solves," it would be helpful to be given more specifics regarding those issues.

4. At best, it seems to assume there are no side effects among the individual assignments, so that order does not matter.

   "It" here is apparently the rewrite algorithm. That algorithm certainly doesn't "assume there are no side effects among the individual assignments." *Au contraire*, in fact: The whole point of that algorithm is precisely to make sure those side effects occur instead of being lost.

5. If this is the assumption, then clearly there are certain ordered sets of assignments (normally coded as transactions) that cannot be rewritten as a multiple assignment because they will produce a result different than that which was originally intended ...

   I can't resist twitting *Critic B* slightly here on his use of the phrase "ordered sets" ... More important, however, we would like to see an example of a sequence of assignments that can't be rewritten as a multiple assignment. The obvious suggestion would be seem to be something along these lines:

   ```
   X := x ;
   Y := f(X) ;
   ```

   But the following multiple assignment will achieve what's presumably intended:

   ```
   X := x ,
   Y := f(x) ;
   ```

6. I like the idea of multiple assignment but not at the expense of transactions and therefore not at the expense of deferred constraint checking.

   We like multiple assignment, too; in fact, we regard it as a sine qua non. Please note, however, that we haven't proposed it as a replacement for transactions. In reference [1], Hugh says the following (and I agree with these remarks):

   I believe that transactions can theoretically be dispensed with but I prefer to keep them for what I believe are strong and possibly compelling reasons of convenience. I know people who disagree with me here and would prefer to get rid of transactions altogether.

I respond to them by agreeing that that might be nice but I need to see some specific language proposals to address the inconvenience that transactions currently address.

DATABASE VALUES AND VARIABLES

Despite everything I've said in this paper so far, there's one sense in which relvars and relational assignment are a mistake after all, as I'll now try to explain.

We want to be able to update the database. Now, I said earlier that "updatable" and "assignable to" mean exactly the same thing; I also said that to be assignable to is to be a variable, and to be a variable is to be assignable to. Doesn't it follow from these remarks that the database is a variable? And since the notion of variables containing variables is a logical absurdity, doesn't it follow further that the database, being a variable, can't possibly contain relation variables?

The answer to both of these questions is in fact *yes:* The database is a variable, and it can't contain other variables (not relation variables and not any other kind) nested inside itself. Here's a quote from Appendix D of reference [14]:

> The first version of *The Third Manifesto* drew a distinction between database values and database variables, analogous to that between relation values and relation variables. It also introduced the term *dbvar* as shorthand for *database variable*. While we still believe this distinction to be a valid one, we found it had little direct relevance to other aspects of the *Manifesto*. We therefore decided, in the interests of familiarity, to revert to more traditional terminology.

After elaborating slightly on these remarks, Appendix D of reference [14] continues:

> Now this bad decision has come home to roost! With hindsight, it would have been much better to "bite the bullet" and adopt the more logically correct terms *database value* and *database variable* (or dbvar), despite their lack of familiarity.

And it goes on to show that (a) a database variable is really a *tuple* variable, with one (relation-valued) attribute for each "relation variable" contained in that database variable; (b) relation variables are really *pseudovariables*, which allow update operations to "zap" individual components of the containing database variable. As Hugh puts it in reference [1]:

> Chris and I contemplated the idea of regarding the database as a single variable [but] we were unable to devise convenient syntax for the usual kinds of ... updating that are expected (assignment of the complete database for every required update being obviously unthinkable). Or rather, the only convenient syntax we could come up with involved dividing the database up into the named "portions" that we call relation variables.

Now, I mention all this merely for completeness and to head off at the pass, as it were, certain criticisms of our position that might occur to some readers. The fact is, even though the database is really a variable and relvars are really pseudovariables, it's my belief that this state of affairs in no way invalidates any of the arguments I've been making earlier in this paper.

CONCLUDING REMARKS

I'd like to conclude with a couple of final observations:

1. First and foremost, the position of *Critics A* and *B* with regard to relvars remains extremely unclear: They seem to think relvars are fundamentally flawed, and yet at the same time they seem to want to retain them, at least "implicitly" (?). They also fail to explain what the logical difference is between a relvar as such and a "time-varying relation."

2. It's true that certain programming languages—specifically, the so-called logic languages (e.g., Prolog) and functional languages (e.g., LISP)—do apparently manage to exist without assignment: indeed, without any notion of "persistent memory" at all. As far as I know, however, all such languages cheat when it comes to updating the database; in effect, they perform some kind of assignment, possibly as a side effect, even though assignment as such isn't part of the logic or functional programming style.

REFERENCES

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

2. E. O. de Brock: "Tables, Table Variables, and Static Integrity Constraints." University of Technology, Eindhoven, Netherlands (1980).

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

4. 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).

5. 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).

6. E. F. Codd: "Extending the Database Relational Model to Capture More Meaning," *ACM TODS 4,* No. 4 (December 1979).

7. E. F. Codd: "Is Your DBMS Really Relational?" (*Computerworld*, October 14th, 1985); "Does Your DBMS Run By The Rules?" (*Computerworld*, October 21st, 1985).

8. E. F. Codd: *The Relational Model for Database Management Version 2*. Reading, Mass.: Addison-Wesley (1990).

9. Hugh Darwen and C. J. Date: *The Third Manifesto. ACM SIGMOD Record 24,* No. 1 (March 1995).

10. C. J. Date: *The Relational Database Dictionary*. Sebastopol, Calif.: O'Reilly Media Inc. (2006, to appear).

11. C. J. Date: "Gödel, Russell, Codd: A Recursive Golden Crowd," *www.thethirdmanifesto.com* (July 2006).

12. C. J. Date: "And Now for Something Completely Computational," *www.thethirdmanifesto.com* (July 2006).

13. C. J. Date and Hugh Darwen: "Multiple Assignment," *www.dbdebunk.com* (February 2004).

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

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