Robert Cummins

Inexplicit Information

The Representation of Knowledge and Belief,
M. Brand and R. M. Harnish, eds. University of Arizona Press, 1986.

Introduction

In a recent conversation with the designer of a chess playing program I heard the following criticism of a rival program: "It thinks it should get its queen out early." This ascribes a propositional attitude to the program in a very useful and predictive way, for as the designer went on to say, one can usually count on chasing that queen around the board. But for all the many levels of explicit representation to be found in that program, nowhere is anything roughly synonymous with "I should get my queen out early" explicitly tokened. The level of analysis to which the designer's remark belongs describes features of the program that are, in an entirely innocent way, emergent properties of the computational processes that have "engineering reality."
-Daniel Dennett

Before discussing the issue raised by this passage, we need to do a little house-cleaning. We are not interested, or shouldn't be, in what representations exist in (are tokened in) the program. Our interest is rather in what representations exist in the system as the result of executing the program, i.e., with representations constructed by the system or that exist in one of its data bases at run time. Thus, although the program Dennett is discussing might contain nothing explicitly about queen deployment, the system might well construct such a representation at run time. For example, the system -- call it CHESS - might begin by executing a rule that says, in effect,

(1) Whenever circumstance C obtains, construct the goal DEPLOY THE CURRENTLY HIGHEST RATED PIECE.

Given the way CHESS typically opens, C regularly obtains early in the game, when the Queen is the highest rated piece. Hence, it typically happens early in the game that the system constructs the goal DEPLOY THE QUEEN.

It is all too easy to write programs that do unintended things like this, so it is all too likely that the case Dennett is actually discussing is like this. But if it is, it is boring. To keep things interesting, I will assume that the system never constructs or uses a representation having the content DEPLOY THE QUEEN. Surely that is the case Dennett intended.

Given that the system never constructs the goal to deploy the queen, what are we to make of the characterization "It thinks it should get its queen out early"? What kind of characterization is that? Dennett's point might be simply that the device behaves *as if* it were executing a program with the explicit goal of early queen deployment. A more interesting interpretation, however, is that information to the effect that the queen should be deployed early is in the system in some sense relevant to the explanation of its

performance, but is not explicitly represented. It is this interpretation I want to pursue, so I'll assume it's what Dennett meant.[1]

What's interesting about this interpretation, of course, is that it assumes the propriety of a kind of intentional characterization – a characterization in terms of propositional content – in the absence of any token in the system having the content in question. I agree with Dennett that it is common and useful to intentionally characterize a system even though the system nowhere explicitly represents the propositional content figuring in the characterization. I have doubts, however, about Dennett's "intentional systems theory" that would have us indulge in such characterization without worrying about *how* the intentional characterizations in question relate to characterizations based on explicit representation. How does "The queen should be deployed early" relate to what is explicitly represented by the system? What determines the content of an intentional characterization at this level of analysis?

In what follows, I will try to distinguish and clarify various distinct types of what I will call *inexplicit information*, i.e., information that exists in a system without benefit of any symbolic structure having the content in question. "Untokened" might be a better term than "inexplicit' for what I have in mind, but it just doesn't sound right. I also have qualms about "information." As Dretske (1981) uses the term, a system cannot have the information that p unless p is the case. I see much merit in this usage, but in the present case it won't do. As I use the term below, information may be false information.

Types of Inexplicit Information

Control-implicit Information

There are many cases in which the natural thing to say is that some piece of information is implicit in the "logic" or "structure" of the flow of control. Imagine a circuit fault diagnosis system so organized that it checks capacitors, if any, only after verifying the power supply, and suppose that it is now executing an instruction like this:

(2) CHECK THE CAPACITOPS.

Given the way the program is structured, the system now has the information that the power supply is OK. It has this information because it cannot execute instruction (2) unless it has verified the power supply. There may be no explicit representation of this fact anywhere in memory -- no token having the content, "The power supply is OK" -- but the fact that control has passed to rule (2) means that the system is in a state that, as Dretske would say, carries the information that the power supply is OK. Programmers, of course, constantly rely on this sort of fact. The system doesn't need to explicitly represent the power supply as being OK because this fact is implicit in the current state of control. This type of implicit information is ubiquitous in almost every computer program.

It isn't difficult to imagine how early queen deployment could be at least partly control implicit in CHESS. The easiest way to see this is to consider how one might

exploit control structure to prevent early deployment of the queen. To imagine a trivial illustration -- this would be a very bad way to build a chess system -- suppose use of the pieces is considered in order, lowest rated pieces being considered first. Only if no acceptable move is found involving a lower rated piece does the system consider moving a higher rated piece. In this way, it might seem, we will avoid queen deployment except when it's really needed. But now suppose that the move evaluation routine was designed with the middle and end game primarily in mind, parts of the game where one might want to rate "aggressiveness" fairly heavily. As a consequence, very few moves in the early part of the game score high, since it is more difficult to make aggressive moves early on. A side effect will be that control will be passed to consideration of queen moves in circumstances that we assumed would occur only in the middle game but which in fact regularly occur earlier. Passing control to a consideration of queen moves amounts to assuming that other moves are insufficiently aggressive, and this assumption will be made much too early in the game.

This sort of possibility comes close to fitting Dennett's original description. Although, in the case imagined, we don't have "Deploy the queen early," we do have the system deciding that only a queen move will do, and doing this early in the game, and doing it without explicitly representing anything like early queen deployment as a goal.

Domain-implicit Information

With a few degenerate exceptions, the information we think of a system as having is always to some extent lodged in the environment. Suppose I write a program, execution of which will get you from your house to mine. Now, in some sense, the program represents me as living in a certain place, perhaps correctly, perhaps not. Where does it say I live? Well, nothing like the proposition

(3) Cummins lives at location L

need be explicitly represented; the program may fix the location in question in that execution of it will get you to my house from yours. But nowhere need there be anything remotely like (3), either in the program itself or constructed at run time.

It is easy to get confused about this and suppose that the location in question must be *inferrable* somehow from the location of your house, together with information given in the program. But this is seriously mistaken. I could give you a perfectly precise program for getting to my house from yours, and another for getting from your house to Paul's, and you could not, without executing them, determine so much as whether Paul and I live in the same place. I could do this by, for example, relying exclusively on LEFT, RIGHT, counting intersections, and counting houses. In such a case, the location of my house just isn't going to be a consequence, in any sense, of premises supplied explicitly. The only way you could use the program to figure out where my house is would be to execute it either in real space or using a sufficiently detailed map. The information in question is as much in the map or geography as it is in the program; the program is completely domain dependent for its success. Nevertheless, given the terrain, the program does carry the

information that my house is at a certain place: if you follow it and wind up at the bus depot, you have every right to complain that I gave you the wrong information.

The phenomenon Dennett describes could be like this. Imagine a set of tactics designed to achieve early control of the center. Suppose they aren't very good; most good opponents can frustrate them, and really good opponents can exploit the situation to draw out the queen, queen deployment being, in the situation that develops, the only way CHESS can protect its knights. Here, the analogue of the geography is the opponents play. It shifts from game to game, but given the way CHESS plays, good opponents are going to respond in similar ways. The resulting interactions typically result in CHESS deploying its queen early. The analogy becomes closer if we imagine that I used *parked cars* as landmarks in my directions. Given that you seldom leave for my house before seven and that most people are home from work by six-thirty and typically park in front of their houses, the result will be that you typically get to my house. But sometimes you will get to the bus depot, or the deli, or somewhere else.

If the phenomenon Dennett describes *is* like this, then it is misleadingly described. If early queen deployment is domain implicit in CHESS, then neither the programmer nor the device executing the program thinks in general, or momentarily, that the queen should be deployed early. Early queen deployment is rather an unintended, unexpected, and perhaps even unnoticed artifact of a variety of factors in the system that have nothing to do with the queen interacting with factors in the "environment" that do. Intuitively, the goal is "in the environment"--i.e., in the opponents play. Indeed, it's the opponent’s goal! Nevertheless, the flaw -- early queen deployment -- is a flaw in the program; it's in the program in just the way that information about the location of my house is in the program consisting of "Two lefts, a right, opposite the blue Chevy station wagon."

Rules, Instructions, and Procedural Knowledge

Once upon a time, there was something called the procedural-declarative controversy in Artificial Intelligence. This controversy had to do with whether it is better to represent knowledge as a procedure -- i.e., as a program applying it -- or as a set of declarative propositions. What everyone decided was that it all depends on what’s convenient for the programming purposes at hand. In short, the controversy died for want of an issue. In a well-known article (*Artificial Intelliqence Meets Natural Stupidity*), Drew McDermott (1976) enjoined us never to talk about it again. Nevertheless, I intend to follow a long philosophical tradition of ignoring the injunctions of such courts and prod these dead issues a little bit anyway.

Let’s begin with an example. Suppose we set out to build a system called FIXIT that diagnoses faults in appliances. Expert system programs -- programs designed to duplicate the performance of an expert in some domain such as appliance failure diagnosis -- are written as sets of *productions*. A production is a rule of the form IF C THEN A, where C is some condition and A is some action. Whenever something in working memory matches C, A is performed. In the simplest case, the rules are unordered; the flow of control is determined solely by which conditions happen to be matched, together with

some simple conflict resolution routines that determine what happens when more than one condition is matched.

The rules of an expert system are supposed to formulate the knowledge of humans who are expert at the task the system is to perform. FIXIT, for example, would probably have a rule like this:

(R) IF APPLIANCE DOESN'T START THEN FIND OUT IF IT IS PLUGGED IN.[2]

Most of *us*, on the other hand, probably begin with the goal to start the appliance, together with a belief like this:

(B) If the appliance isn't plugged in, then it won't start.

If the appliance doesn't start, we use (B) and some inference procedures to construct a subgoal: find out if the appliance is plugged in. Experts, unlike the rest of us, seem to have "proceduralized" this business: they just execute (R). The difference is that novices must remember that an appliance won't start if it is unplugged and then reason from this to the conclusion that the plug should be checked. Experts don't have to figure out what to do: they simply check the plug when the thing won't start.

There are two ways in which a system can "have" a rule like (R): (R) might be a rule that is represented in the system's memory, or (R) might be an *instruction*, i.e., a rule in the program that the system executes. These are quite different matters. Let's take them in order.[3]

First, then, suppose that FIXIT has rule (R) represented in memory. (R), then, is one of the rules it "knows". But having access to (R) is evidently quite a different matter from having access to (B). A system that has access to (R) knows what to do if the appliance won't start. A system with access to (B) must *infer* what to do and hence must have some capacity for means-ends analysis. This is why a system operating on the basis of (R) can be expected to make different sorts of errors than a system operating on the basis (B), and why a system operating on the basis of (B) can be expected to be slower than one operating on the basis of (R). It is precisely because a system with access to (R) doesn't need to infer what to do that it is a mistake to suppose its access to (R) amounts to knowing the same thing it would know if it had access to (B) instead. A system with access to (R) knows what to do, and a system with access to (B) instead does not: it must figure out what to do.[4]

So much, then, for the case in which (R) is represented in memory -- a genuine rule. What about the case in which (R) is a rule in the program that the system executes -- i.e., an instruction? Let's move back to the original example derived from Dennett. Suppose the *program* executed by the device CHESS contains the following rule:

IF IT IS EARLY IN THE GAME THEN DEPLOY THE QUEEN.

Does CHESS -- the *device* executing the program -- believe that it should deploy its queen early? The programmer certainly believed it. And CHESS will behave as if it believed it too: hence the characterization Dennett reports. But CHESS (as we are now imagining it) simply executes the rule without representing it at all, except in the degenerate sense in which rubber bands represent the rude IF PULLED THEN STRETCH, and masses represent the rule COALESCE WITH OTHER MASSES. Like the rubber band, CHESS simply executes its rule, and executing that rule amounts to having a behavioral disposition to deploy the queen early.[5] By contrast, it does represent the state of the game -- e.g., the positions of the pieces and the fact that it is early in the game --and it does that by executing a set of instructions that say, in effect, REPRESENT THE CURRENT STATE OF THE GAME. Moreover, the system not only represents facts about the game, but its representations of these facts play something like the role that functionally distinguishes belief from other intentional states, viz., availability as premises in reasoning and susceptibility to evidential evaluation (if CHESS is a learner). But CHESS (as we are now imagining it) does not represent the rule requiring early queen deployment, nor is anything with a comparable content available for reasoning or epistemic assessment. Consequently, I think we should resist the claim that CHESS thinks or believes or knows that it should deploy the queen early. The rules CHESS executes -- what I've been calling its instructions -- are quite different and have a very different explanatory role from the rules CHESS knows.[6]

A frequent reply to this point is that our system has procedural knowledge to the effect that the queen should be deployed early in virtue of executing a program containing the rule. Ordinary usage condones this line to some extent by allowing us to describe the capacities of cognitive systems as knowing how to do something even though there is no explicit tokening of the rules executed. Notice, however, that we aren't allowed this license when speaking of noncognitive systems: rubber bands don't know how to stretch when pulled. I think talk of procedural knowledge has its place -- it's the case we discussed a moment ago in which we have (R) explicitly tokened in memory -- but a system's procedural knowledge is not knowledge of the rules it executes. The rules a system executes -- the ones making up its program -- are not available for reasoning or evidential assessment for the simple reason that they are not represented to the system at all.[7]

Of course, if CHESS is implemented on a general purpose computer, rather than hardwired, the program itself will also be represented "in" the system: it may be, on a disk, for example. But these are not representations to CHESS, they are representations to the system that implements CHESS, typically an interpreter and operating system. The interpreter “reads" these rules, not CHESS. The program file is not a data base for CHESS it is a data base for the interpreter. CHESS doesn't represent its program, it executes it, and this is made possible by the fact that a quite different system *does* represent CHESS’s program. If we hardwire CHESS, the program is no longer represented at all; it is merely "embodied." There is all the difference in the world between, writing a program that has *access* to a rule codified in one of its data structures and a program that *contains* that rule as an instruction.[8]

The presence of our rule in CHESS's program, therefore, indicates something about the programmer's knowledge but nothing one way or the other about CHESS's representational states. Contrast a case that *does* tell us something about CHESS's representational states:

IF OPPONENT HAS MOVED THEN UPDATE THE CURRENT POSITION.

When this instruction is executed, CHESS will create a representation of the current position and store it for future access.

Once we are clear about the distinction between representing a rule and executing it, we are forced to realize that production systems demonstrate that a system can have a cognitive skill or ability -- e.g., the ability to diagnose appliance failure or to play chess -- without knowing the sorts of things appliance fixers or chess players typically know. When knowledge is "proceduralized," it ceases to be knowledge, if by "proceduralization" we mean that the rules in question become instructions in the program the system executes. Nevertheless, and here is the main point at last, even though the system doesn't represent such rules, the fact that it executes them amounts to the presence in the system of some propositionally formulatable information, information that is not explicitly represented but is inexplicit in the system in virtue of the physical structure upon which program execution supervenes.

If we turn back to Dennett's parable now, we find something of a muddle. Evidently, the case in which we have a rule explicitly tokened in memory is not the one at issue. In the case lately imagined, however, although we do have an instruction *in the program*, nothing like DEPLOY THE QUEEN EARLY is tokened *by the system*. This is the case we are interested in, but here it seems plainly incorrect to say that the system thinks or believes that it should deploy the queen early, though this does apply to the programmer. Instead, the correct description seems to be that in the system there is a kind of inexplicit information, information lodged, as it were, in whatever physical facts underwrite the capacity to execute the program. Explanatory appeals to this sort of information evidently differ radically from explanatory appeals to the system's representations -- e.g., representations of current position. Moreover, it is plain that it is the appeal to rules *executed* -- i.e., to instructions -- that is the basic explanatory appeal of cognitive science. It is only in virtue of the instructions a system executes that the knowledge it has can issue in behavior. Indeed, it is only because of the instructions executed that it has any representational states at all.

Conclusion

We have now seen several ways in which it makes a kind of sense to describe a chess system in terms of informational contents that are not explicitly represented. Moreover, each type of inexplicit representation appears to have a, bona fide -- indeed essential -- explanatory role, and, though the details want spelling out, each seems to supervene in conceptually straight-forward ways on ontologically kosher features of program-executing systems. In general, once we realize that a system can have all kinds of

information that isn't in its memory, information that *it* does not represent at all, we see that intentional characterization -- characterization in terms of informational content -- is not all of a piece and that the different pieces have radically different explanatory roles. Explanation by appeal to any of the three kinds of inexplicit information I have been discussing is always explanation by appeal to rules executed and hence is quite different from explanation by appeal to knowledge structures. Everyone realizes that what you can do is a function of what information you have, but not everyone in cognitive science seems to realize the importance of information that is not explicitly represented or stored in memory.

I've been making much of the distinction between representing a rule and executing it. Executing a rule, I've been urging, isn't knowing or believing it. And conversely, there is ample evidence that knowing a rule isn't sufficient for being able to execute it. Explicit information -- knowledge and belief properly so-called -- is a matter of which representations are created or exist in the system at run time and of how these are used by the system. If a representation of the world exists in the system, is available as a premise in reasoning, and is subject to evidential assessment, we have at least the salient *necessary* conditions for intentional characterization as knowledge or belief. *Inexplicit* information, on the other hand, is a matter of which rules or instructions the system executes -- a matter of its program -- and the environment the system operates in. A system can have much information, that is *not represented by the system at all* and that doesn't function anything like knowledge or belief, even tacit knowledge or belief. When we formulate the content of this information and attribute it to the system, we intentionally characterize that system, and rightly so, even though the propositional contents of our characterizations are not represented by the system we characterize. But we are not characterizing what it knows or believes.

Current cognitive science, with its emphasis -- nay, fixation -- on "knowledge representation," has neglected (officially, if not in practice) the ubiquitous and critical information that isn't represented[9] (not represented by the cognitive system anyway). It *is* represented by the programmer, of course, and one of the important insights of the cognitive science movement is that a program is a theory. When we write a program, we are theorizing about our human subjects, representing to each other the rules we suppose they execute. When we do this, what we are doing to a great extent is specifying the inexplicit information that drives human as well as computer cognitive processes. Everyone recognizes the importance of what is represented. I've been urging the importance of what isn't.

---

[1] Perhaps Dennett doesn't distinguish these two interpretations being an instrumentalist about goals. But he should, as the sequel will show, I think.

2 Something must be done, of course, to prevent the left-hand side of this rude matching forever. The completed rule might read: IF A DOESN'T START AND NOT (PLUG CHECKED) THEN CHECK PLUG AND WRITE (PLUG CHECKED).

[3] The literature on expert systems treats this distinction with a benign neglect. It only matters if we want the system to be able to alter its own productions, in which case they must be rules, not instructions.

[4] The received view is that we have the *same knowledge* represented in each case, but represented in different forms -- a declarative form and a procedural form -- and this difference is held to account for the characteristic differences in performance. But is this right? Is the difference only a difference in how the

same thing is represented? It seems clear the difference is a difference in what is known rather than simply a difference in how what is known is represented. To make the point once again: the system with (R) knows what to do, whereas the other must figure it out.
(Added later: It is an important Kantian insight that representing something in a different form always changes what is represented.)

[5] We might mark the distinction by saying that instructions are "embodied" in the device that executes them. Cf. the notion of E-representation 'in Cummins 1977,1983.
[6] For more on the explanatory role of appeals to instructions executed see Cummins 1977, 1983.
[7] When inference-dependent propositions like (B) are replaced by rules like (R) *in memory*, we have a kind of proceduralization that does yield knowledge but not the *same* knowledge.
[8] From an Al perspective, this is trivial, of course: simply putting a rule in memory is not going to get it executed; we must somehow pass control to it.
[9] "Knowledge representation" seems a misnomer: What we are interested in is how a cognitive system represents (or should represent) the world, not how to represent knowledge. 'How does S represent the world?" = "How is S's knowledge *encoded*?"

Cognitive science might be interested in knowledge representation in the following sense: how should we, as theorists, represent in our theoretical notation the knowledge a cognitive system has of say, chess. This is a legitimate issue but not the one generally meant by "knowledge representation."