Understanding Computers and Cognition

A New Foundation for Design

TERRY WINOGRAD
FERNANDO FLORES

"This is a ground breaking book. It is about computers, artificial intelligence, language . . .

Chapter 7

Computers and representation

This book is directed towards understanding what can be done with computers. In Part I we developed a theoretical orientation towards human thought and language, which serves as the background for our analysis of the technological potential. In Part II we turn towards the technology itself, with particular attention to revealing the assumptions underlying its development. In this chapter we first establish a context for talking about computers and programming in general, laying out some basic issues that apply to all programs, including the artificial intelligence work that we will describe in subsequent chapters. We go into some detail here so that readers not familiar with the design of computer systems will have a clearer perspective both on the wealth of detail and on the broad relevance of a few general principles.

Many books on computers and their implications begin with a description of the formal aspects of computing, such as binary numbers, Boolean logic, and Turing machines. This sort of material is necessary for technical mastery and can be useful in dispelling the mysteries of how a machine can do computation at all. But it turns attention away from the more significant aspects of computer systems that arise from their larger-scale organization as collections of interacting components (both physical and computational) based on a formalization of some aspect of the world. In this chapter we concentrate on the fundamental issues of language and rationality that are the background for designing and programming computers.

We must keep in mind that our description is based on an idealization in which we take for granted the functioning of computer systems according to their intended design. In the actual use of computers there is a critical larger domain, in which new issues arise from the breakdowns ('bugs' and 'malfunctions') of both hardware and software. Furthermore, behind these technical aspects are the concerns of the people who design, build, and use the devices. An understanding of what a computer really does is an understanding of the social and political situation in which it is designed, built, purchased, installed, and used. Most unsuccessful computing systems have been relatively successful at the raw technical level but failed because of not dealing with breakdowns and not being designed appropriately for the context in which they were to be operated.[1]

It is beyond the scope of our book to deal thoroughly with all of these matters. Our task is to provide a theoretical orientation within which we can identify significant concerns and ask appropriate questions. In showing how programming depends on representation we are laying one cornerstone for the understanding of programs, and in particular of programs that are claimed to be intelligent.

7.1 Programming as representation

The first and most obvious point is that whenever someone writes a program, it is a program about something.[2] Whether it be the orbits of a satellite, the bills and payroll of a corporation, or the movement of spaceships on a video screen, there is some subject domain to which the programmer addresses the program.

For the moment (until we refine this view in section 7.2) we can regard the underlying machine as providing a set of storage cells, each of which can hold a *symbol structure*, either a number or a sequence of characters (letters, numerals, and punctuation marks). The steps of a program specify operations on the contents of those cells—copying them into other cells, comparing them, and modifying them (for example by adding two numbers or removing a character from a sequence).

In setting up a program, the programmer has in mind a systematic correspondence by which the contents of certain storage cells *represent* objects and relationships within the subject domain. For example, the contents of three of the cells may represent the location of some physical object with respect to a Cartesian coordinate system and unit of measurement. The operations by which these contents are modified as the program runs are designed to correspond to some desired calculation about the location of

[1] The nature and importance of this social embedding of computers is described by Kling and Scacchi in "The web of computing"(1982).

[2] We will ignore special cases like the construction of a sequence of instructions whose purpose is simply to exercise the machine to test it for flaws.

that object, for example in tracking a satellite. Similarly, the sequence of characters in a cell may represent the name or address of a person for whom a paycheck is being prepared.

Success in programming depends on designing a representation and set of operations that are both *veridical* and *effective*. They are veridical to the extent that they produce results that are correct relative to the domain: they give the actual location of the satellite or the legal deductions from the paycheck. They are effective to varying degrees, depending on how efficiently the computational operations can be carried out. Much of the detailed content of computer science lies in the design of representations that make it possible to carry out some class of operations efficiently.

mostly in CS...

YES; TC is somewhat differ

Research on artificial intelligence has emphasized the problem of representation. In typical artificial intelligence programs, there is a more complex correspondence between what is to be represented and the corresponding form in the machine. For example, to represent the fact that the location of a particular object is "between 3 and 5 miles away" or "somewhere near the orbiter," we cannot use a simple number. There must be conventions by which some structures (e.g., sequences of characters) correspond to such facts. Straightforward mappings (such as simply storing English sentences) raise insuperable problems of effectiveness. The operations for coming to a conclusion are no longer the well-understood operations of arithmetic, but call for some kind of higher-level reasoning.

In general, artificial intelligence researchers make use of formal logical systems (such as predicate calculus) for which the available operations and their consequences are well understood. They set up correspondences between formulas in such a system and the things being represented in such a way that the operations achieve the desired veridicality. There is a great deal of argument as to the most important properties of such a formal system, but the assumptions that underlie all of the standard approaches can be summarized as follows:

1. There is a structure of formal symbols that can be manipulated according to a precisely defined and well-understood system of rules.

2. There is a mapping through which the relevant properties of the domain can be represented by symbol structures. This mapping is systematic in that a community of programmers can agree as to what a given structure represents. *

3. There are operations that manipulate the symbols in such a way as to produce veridical results—to derive new structures that represent the domain in such a way that the programmers would find them accurate representations. Programs can be written that combine these operations to produce desired results.

The problem is that representation is in the mind of the beholder. There is nothing in the design of the machine or the operation of the program that depends in any way on the fact that the symbol structures are viewed as representing anything at all.[3]

There are two cases in which it is not immediately obvious that the significance of what is stored in the machine is externally attributed: the case of robot-like machines with sensors and effectors operating in the physical world, and the case of symbols with internal referents, such as those representing locations and instructions within the machine. We will discuss the significance of robots in Chapter 8, and for the moment will simply state that, for the kinds of robots that are constructed in artificial intelligence, none of the significant issues differ from those discussed here.

The problem of 'meta-reference' is more complex. Newell and Simon, in their discussion of physical symbol systems ("Computer science as an empirical inquiry," 1976), argue that one essential feature of intelligent systems is that some of the symbols can be taken as referring to operations and other symbols within the machine: not just for an outside observer, but as part of the causal mechanism.

Even in this case there is a deep and important sense in which the referential relationship is still not intrinsic. However, the arguments are complex and not central to our discussion. We are primarily concerned with how computers are used in a practical context, where the central issue is the representation of the external world. The ability of computers to coherently represent their own instructions and internal structure is an interesting and important technical consideration, but not one that affects our perspective.

7.2 Levels of representation

In the previous section, computers were described rather loosely as being able to carry out operations on symbol structures of various kinds. However this is not a direct description of their physical structure and functioning. Theoretically, one could describe the operation of a digital computer purely in terms of electrical impulses travelling through a complex network of electronic elements, without treating these impulses as symbols for anything. Just as a particular number in the computer might represent some relevant domain object (such as the location of a satellite), a deeper analysis shows that the number itself is not an object in

[3] This point has been raised by a number of philosophers, such as Fodor in "Methodological solipsism considered as a research strategy in cognitive psychology" (1980), and Searle in "Minds, brains, and programs" (1980). We will discuss its relevance to language understanding in Chapter 9.

the computer, but that some pattern of impulses or electrical states in turn *represents* the number. One of the properties unique to the digital computer is the possibility of constructing systems that cascade levels of representation one on top of another to great depth.

The computer programmer or theorist does not begin with a view of the computer as a physical machine with which he or she interacts, but as an abstraction—a formalism for describing patterns of behavior. In programming, we begin with a language whose individual components describe simple acts and objects. Using this language, we build up descriptions of *algorithms* for carrying out a desired task. As a programmer, one views the behavior of the system as being totally determined by the program. The language implementation is opaque in that the detailed structure of computer systems that actually carry out the task are not relevant in the domain of behavior considered by the programmer.

If we observe a computer running a typical artificial intelligence program, we can analyze its behavior at any of the following levels:

The physical machine. The machine is a complex network of components such as wires, integrated circuits, and magnetic disks. These components operate according to the laws of physics, generating patterns of electrical and magnetic activity. Of course, any understandable description will be based on finding a modular decomposition of the whole machine into components, each of which can be described in terms of its internal structure and its interaction with other components. This decomposition is recursive—a single component of one structure is in turn a composite made up of smaller structures. At the bottom of this decomposition one finds the basic physical elements, such as strands of copper and areas of semiconductor metal laid down on a wafer of silicon crystal. It is important to distinguish this kind of hierarchical decomposition into components (at a single level) from the analysis of levels of representation.

The logical machine. The computer designer does not generally begin with a concept of the machine as a collection of physical components, but as a collection of logical elements. The components at this level are logical abstractions such as or-gates, inverters, and flip-flops (or, on a higher level of the decomposition, multiplexers, arithmetic-logical units, and address decoders). These abstractions are represented by activity in the physical components. For example, certain ranges of voltages are interpreted as representing a logical 'true' and other ranges a logical 'false.' The course of changes over time is interpreted as a sequence of discrete cycles, with the activity considered stable at the end of each cycle. If the machine is properly designed, the representation at this level is veridical—patterns of activity interpreted as logic will lead to other patterns according to the rules of logic. In any real machine, at early stages of debugging, this representation will be incomplete. There will be behavior caused by phenomena such as irregular voltages and faulty synchronization that does not accurately represent the logical machine. In a properly working machine, all of the *relevant* physical behavior can be characterized in terms of the logic it represents.

The abstract machine. The logical machine is still a network of components, with activity distributed throughout. Most of today's computers are described in terms of an abstract single sequential processor, which steps through a series of instructions. It is at this level of representation that a logical pattern (a pattern of trues and falses) is interpreted as representing a higher-level symbol such as a number or a character. Each instruction is a simple operation of fetching or storing a symbol or performing a logical or arithmetic operation, such as a comparison, an addition, or a multiplication. The activity of the logical machine cannot be segmented into disjoint time slices that represent the steps of the abstract machine. In a modern machine, at any one moment the logical circuits will be simultaneously completing one step (storing away its results), carrying out the following one (e.g., doing an arithmetic operation), and beginning the next (analyzing it to see where its data are to be fetched from). Other parts of the circuitry may be performing tasks needed for the ongoing function of the machine (e.g., sending signals that prevent items from fading from memory cells), which are independent of the abstract machine steps. Most descriptions of computers are at the level of the abstract machine, since this is usually the lowest level at which the programmer has control over the details of activity.[4]

A high-level language. Most programs today are written in languages such as FORTRAN, BASIC, COBOL, and LISP, which provide elementary operations at a level more suitable for representing real-world domains. For example, a single step can convey a complex mathematical operation such as "x = (y+z)*3/z." A compiler or interpreter[5] converts a formula like this into a sequence of operations for the abstract machine. A higher-level language can be based on more complex symbol structures, such as lists, trees, and character strings. In LISP, for example, the contents of a number of storage cells in the underlying abstract machine can

[4] Even this story is too simple. It was true of computers ten years ago, but most present-day computers have an additional level called 'micro-code' which implements the abstract machine instructions in terms of instructions for a simpler abstract machine which in turn is defined in terms of the logical machine.

[5] The difference between compiling and interpretation is subtle and is not critical for our discussion.

be interpreted together as representing a list of items. To the LISP programmer, the list "(APPLES ORANGES PUDDING PIE)" is a single symbol structure to which operations such as "REVERSE" can be applied. Once again, there need be no simple correspondence between an operation at the higher level and those at the lower level that represent it. If several formulas all contain the term "(y+z)" the compiler may produce a sequence of machine steps which does the addition only once, then saves the result for use in all of the steps containing those formulas. If asked the question "Which formula is it computing right now?" the answer may not be a single high-level step.

A representation scheme for 'facts'. Programs for artificial intelligence use the symbol structures of a higher-level language to represent facts about the world. As mentioned above, there are a number of different conventions for doing this, but for any one program there must be a uniform organization. For example, an operation that a programmer would describe as "Store the fact that the person named 'Eric' lives in Chicago" may be encoded in the high-level language as a series of manipulations on a data base, or as the addition of a new proposition to a collection of axioms. There will be specific numbers or sequences of characters associated with "Eric" and "Chicago" and with the relationship "lives in." There will be a convention for organizing these to systematically represent the fact that it is Eric who lives in Chicago, not vice versa. At this level, the objects being manipulated lie once again in the domain of logic (as they did several levels below), but here instead of simple Boolean (two-valued) variables, they are formulas that stand for propositions. The relevant operators are those of logical inference, such as instantiating a general proposition for a particular individual, or using an inference rule to derive a new proposition from existing ones.

In designing a program to carry out some task, the programmer thinks in terms of the subject domain and the highest of these levels that exists for the programming system, dealing with the objects and operations it makes available. The fact that these are in turn represented at a lower level (and that in turn at a still lower one) is only of secondary relevance, as discussed in the following section. For someone designing a program or piece of hardware at one of the lower levels, the subject domain is the next higher level itself.

The exact form of this tower of levels is not critical, and may well change as new kinds of hardware are designed and as new programming concepts evolve. This detail has been presented to give some sense of the complexity that lies between an operation that a programmer would mention in describing what a program does and the operation of the physical computing device. People who have not programmed computers have not generally had experiences that provide similar intuitions about systems. One obvious fact is that for a typical complex computer program, there is no intelligible correspondence between operations at distant levels. If you ask someone to characterize the activity in the physical circuits when the program is deciding where the satellite is, there is no answer that can be given except by building up the description level by level. Furthermore, in going from level to level there is no preservation of modularity. A single high-level language step (which is chosen from many different types available) may compile into code using all of the different machine instructions, and furthermore the determination of what it compiles into will depend on global properties of the higher-level code.

7.3 Can computers do more than you tell them to do?

Readers who have had experience with computers will have noted that the story told in the previous section is too simple. It emphasizes the opacity of implementation, which is one of the key intellectual contributions of computer science. In the construction of physical systems, it is a rare exception for there to be a complete coherent level of design at which considerations of physical implementation at a lower level are irrelevant. Computer systems on the other hand can exhibit many levels of representation, each of which is understood independently of those below it. One designs an algorithm as a collection of commands for manipulating logical formulas, and can understand its behavior without any notion of how this description will be written in a higher-level language, how that program will be converted into a sequence of instructions for the abstract machine, how those will be interpreted as sequences of instructions in micro-code, how those in turn cause the switching of logic circuits, or how those are implemented using physical properties of electronic components. Theoretically, the machine as structured at any one of these levels could be replaced by a totally different one without affecting the behavior as seen at any higher level.

We have oversimplified matters, however, by saying that all of the relevant aspects of what is happening at one level can be characterized in terms of what they represent at the next higher level. This does not take into account several issues:

Breakdowns. First of all, the purely layered account above is based on the assumption that each level operates as a representation exactly as anticipated. This is rarely the case. In describing the step from electronic circuits to logic circuits, we pointed out that it took careful debugging to guarantee that the behavior of the machine could be accurately described in terms of the logic. There is a similar problem at each juncture, and a person writing a program at any one level often needs to understand (and potentially modify) how it is represented at the one below. The domain of breakdowns generated by the lower levels must be reflected in the domain for understanding the higher ones. This kind of interdependence is universally viewed as a defect in the system, and great pains are taken to avoid it, but it can never be avoided completely.

Resource use. Even assuming that a description at a higher level is adequate (the representation is veridical), there may be properties of the machine that can be described only at a lower level but which are relevant to the efficiency with which the higher-level operations are carried out. For example, two operations that are both primitive in a higher-level language may take very different amounts of time or physical storage to run on a given machine with a given *implementation* (representation of the higher-level language on the abstract machine). Although this may not be relevant in specifying what the result will be, it will be relevant to the process of getting it. In real-time systems, where the computer activates physical devices at times that have relevance in the subject domain (e.g., a controller for an industrial process, or a collision avoidance system for aircraft), the speed of execution may be critical. In the use of storage, there are often limits on how much can be stored, and the details of when these limits will be reached can be described only on the lower levels.

Differing attitudes are taken to cross-level dependencies that deal with resources. Some programmers argue that whenever resources are significant, the program should be written at the level where they can be directly described, rather than a higher level. For example they argue that real-time control processes should be written in assembly language (a language that corresponds closely to the abstract machine) rather than in a higher-level language, since the resources connected with the objects and operations of the abstract machine can be directly specified. Others argue that the program should be designed at the higher level only, and that the lower-level systems should provide higher-level operations that are so efficient that there never need be a concern. In practice, programs are often initially designed without taking into account the lower level, and then modified to improve performance.

Accidental representation. There are some cases in which there are useful higher-level descriptions of a program's behavior that do not correspond to an intentional representation by a programmer. As a simple example, there have been a number of 'display hack' programs that produce geometrical designs on a computer screen. Many of these grew out of programs that were originally created to perform operations on symbols that had nothing to do with visual figures. When the contents of some of their internal storage cells were interpreted as numbers representing points on a video display screen, strikingly regular patterns emerged, which the programmer had not anticipated. One such program produces figures containing circular forms and might be appropriately described as "drawing a circle," even though the concept of circle did not play a role in the design of its mechanisms at any level. In these cases, the description of the program as representing something is a description by some observer after the fact, rather than by its designer.

If it were not for this last possibility we could argue that any properly constructed computer program is related to a subject domain only through the relationships of representation intended by its programmers. However there remains the logical possibility that a computer could end up operating successfully within a domain totally unintended by its designers or the programmers who constructed its programs.

This possibility is related to the issues of structural coupling and instructional interaction raised by Maturana. He argues that structures in the nervous system do not represent the world in which the organism lives. Similarly one could say of the display hack program that its structures do not represent the geometrical objects that it draws. It is possible that we might (either accidentally or intentionally) endow a machine with essential qualities we do not anticipate. In Section 8.4 we will discuss the relevance of this observation to the question of whether computers can think.