IMPLEMENTATION IS SEMANTIC INTERPRETATION

William J. Rapaport

Department of Computer Science, Department of Philosophy, and Center for Cognitive Science
State University of New York at Buffalo, Buffalo, NY 14260

`rapaport@cs.buffalo.edu`
`http://www.cs.buffalo.edu/pub/WWW/faculty/rapaport/`

Abstract

What is the computational notion of "implementation"? It is not individuation, instantiation, reduction, or supervenience. It is, I suggest, semantic interpretation.

This document is *Technical Report 97-15* (Buffalo: SUNY Buffalo Department of Computer Science) and *Technical Report 97-5* (Buffalo: SUNY Buffalo Center for Cognitive Science).

1 INTRODUCTION

Consider the relationships among algorithms, computer programs, and the computers that execute them. An algorithm is (roughly) a procedure for computing a function (for more details, see Soare 1996; Rapaport, forthcoming). A program is a more specific and detailed *textual* expression of an algorithm, expressed in a programming language. Often, it is said that the program "implements" the algorithm. A computer *process* is an algorithm being executed (see Rapaport 1988, 1995; Smith 1997). It is a physical device (a computer) *behaving* in a certain *way*; the *way* is described (or specified) by the program, and the physical device running the process *implements* the program. But what is "implementation"? In this paper, I present evidence that *implementation is semantic interpretation.*

Semantic interpretation requires two domains and one relation: a *syntactic domain*, characterized by rules of "symbol manipulation" (perhaps suitably generalized to be able to deal with domains that are not strictly speaking "symbolic"); a *semantic domain*, similarly characterized; and a relation of semantic interpretation that maps the former to the latter (cf. Rapaport 1988, 1995). Put this way, there is no *intrinsic* difference between the domanins; what makes one syntactic and the other semantic is the asymmetry of the interpretation mapping. Thus, a given domain can be either syntactic *or* semantic, depending on the other domain. E.g., a computer process that implements a program plays the role of semantic domain to the program's role as syntactic domain. The same program, implementing an algorithm, plays the role of semantic domain to the algorithm's role as the syntactic domain. The thesis to be explicated and justified is (roughly) that a semantic domain *implements* a syntactic domain. For reasons that will become clear below, I shall use the term 'Abstraction' for the syntactic domain; thus, an implementation is a semantic interpretation of an Abstraction.

semantic domain		syntactic domain
1. a computer program	is an implementation of	an algorithm
2. a computational process	is an implementation of	a computer program
3. a data structure	is an implementation of	an abstract data type
4. a performance	is an implementation of	a musical score or play-script
5. a house	is an implementation of	a blueprint
6. a set-theoretic model	is an implementation of	a formal theory

Table 1: Semantic domains that are implementations of syntactic domains.

Table 1 shows pairs of syntactic and semantic domains that are clear examples of implementations (cf. Rapaport 1995 for a more elaborate survey). The first three are paradigmatic cases: We implement an algorithm when we express it in a computer programming language; we implement a program when we compile and execute it; and we implement an abstract data type such as a stack when we write code (in some programming language) that specifies *how* the various stack operations (such as *push* and *pop*) will work. Cases 4 and 5 are clearly of the same type as these paradigms, even though we don't, normally, use the term 'implementation' in discussing them. Case 6, another example that arguably can be thought of in the same way, suggests, in addition, that all semantic interpretations can be seen as implementations.

2 IMPLEMENTATION IN COMPUTER SCIENCE.

Although the term 'implementation' antedates computer science and is used elsewhere, it is ubiquitous in computer science, so let's begin our investigation there.

Given the ubiquity, it is rather surprising how few texts even try to define the notion. For instance, all that a standard text on programming languages says is that "the **realization** of a programming language in a computer system is called the *implementation* (Marcotty & Ledgard 1986: 8; my boldface). 'Realization' is left undefined. Taken literally, it means "to make real", where 'real' is opposed to 'imaginary' or perhaps 'abstract'. This makes it seem that the physical medium is important and that to "realize" X could be to establish a real-world, physical correlate of X.[1]

"Realization" itself is an interesting notion. According to the new *Oxford English Dictionary*, 'real' comes from the Latin for "pertaining to things", and its philosophical meaning, in part, is "having an existence in fact and not merely in appearance, thought, or language" (Simpson & Weiner 1989, 13: 272). What is made real when it is "realized"? Presumably, something that exists "merely in appearance, thought, or language"—something that is syntactically characterized or "Abstract". To *realize* is, in part, "To make real, to give reality to (something merely imagined, planned, etc.) ... In common use from *c* 1750 with a variety of objects, as ideas or ideals, schemes, theories, hopes, fears, etc. ..." (Simpson & Weiner 1989, 13: 277). Note how *psychological* or *intentional* these realizable things are.

2.1 Implementation and the Semantic "Gap".

Another computer-science text says a bit more about implementation (Hayes 1988: 47; my boldface):

> With the advent of the [IBM] System/360, the distinction between a computer's architecture and its **implementation** became apparent. As defined by the System/360 designers ..., the *architecture* of a computer is its **structure and behavior as seen by an assembly-language programmer** .... The *implementation* ... refers to the **logical and physical design techniques** used to **realize** the architecture in any specific instance. Thus all the members of the S/360–370 series share a common architecture, but they have many different implementations. For example, some S/360–370 CPUs employ fast hardwired control units, whereas others use a slower but more flexible microprogrammed approach to implementing the common instruction set.

Architecture is concerned with structure and behavior; these are functional, "Abstract" aspects. This is not to say that it is not *detailed*, however, since the architecture is "seen by an assembly-language programmer," who must know all about the details of registers, control, etc., although without having to worry about what a register looks like or how the control is actually carried out. *Implementation* is concerned with "logical and physical design techniques". 'Logical' here probably refers to "logic" gates, which are themselves physical. Thus, implementations are the physical realizations of "Abstractions".

I would call a *physical* realization a special case, however. The full explication of 'implementation' requires a third term besides the implementation and the Abstraction; the relation is ternary:

I is an implementation of A in medium M

where A is the Abstraction and M could be physical or set-theoretical, etc. For instance, in the study of data structures, one talks about implementing a stack by means of a linked list, implementing the list in a programming language (say, Pascal), "implementing"—i.e., compiling—the Pascal program in some machine language (ML),[2] and then implementing the ML program in a real computer. As we progress along this sequence (this "correspondence continuum"; cf. Smith 1987, Rapaport 1995), the implementing media begin as Abstractions themselves and gradually take on a more "physical" nature. Perhaps you will object that program compilation should not be treated as an implementation. Hayes, however, would not object: "a sequence of ... [machine] instructions is needed to *implement* a statement in a high-level programming language such as FORTRAN or Pascal" (Hayes 1988: 209, my italics). So, to implement is to "realize" *in* some medium, which might be a physical medium or could be some *domain* or *language*.

To implement is to *construct* something, out of the materials at hand, that has the properties of the Abstraction; it could also be to *find* a counterpart that has those properties. *Both tasks are semantic.* Hayes, indeed, speaks of semantics in this context (Hayes 1988: 209; my boldface):

> Because of the complexity of the operations, data types, and syntax of high-level languages, few successful attempts have been made to construct computers whose machine language directly corresponds to a high-level language .... There is thus a *semantic gap* between the high-level problem specification and the machine instruction set that **implements** it, a gap that a compiler must bridge.

What is this gap? Presumably, that (say) the specific operations, data types, etc., of the high-level language don't correspond directly to anything in the machine language: Pascal, e.g., has the "record" data type, but my Sun's machine language probably doesn't. So, a compiler is needed to show how to construct or implement records in the machine language.

Why does Hayes call this a *semantic* gap? It's a bit like the fact that one natural language might not have a single word corresponding to some single word in another natural language. Consider Russian, which has a term, 'ruka', referring to what in English has to be referred to as the hand+forearm.[3] Of course, one can translate between the languages by defining the word in terms of others (perhaps with a cultural gloss; cf. Jennings 1985, Rapaport 1988: 102). But why is this *semantic* rather than *syntactic*?

A possible interpretation[4] of the "semantic gap" specifies four relations:

A. A program in a high-level programming language is semantically interpreted by real-world objects. Presumably, the semantic interpretation of the program is the relation between, on the one hand, data structures (say) in a Pascal program (e.g., a record representing a student viewed as consisting of a name, a class, a major, a student-number, and a grade-point average) and, on the other hand, an actual student in the real world.

B. The program is also compiled into an ML implementation. The compilation relation is, or includes, the relation between that student-record data structure and a construct of data types in the machine language. Both A and B are semantic relations.

C. The ML implementation, in turn, is semantically interpreted by bits in a computer. It may seem odd to semantically interpret the ML program by bits, rather than by the real-world objects. However, as I argue elsewhere (Rapaport 1995), all semantic relations are merely correspondences (and vice versa). Thus, this relation between the ML program and bits is just another correspondence. After all, we could also have mapped the Pascal program into computer bits—in fact, via B and C, we have! So, an ML program can be interpreted in terms of bits in the computer. Arguably, in fact, having these two distinct interpretations of two distinct (albeit input-output-equivalent) programs is appropriate. Where the ML program talks of registers, the Pascal program talks of "students" (or student-records). So it is appropriate to understand the Pascal program as a "mathematical model" of such real-world objects as students, and to understand the ML program as a "mathematical model" of such (also real-world) objects as bits in a computer.

D. The semantic gap concerns the relation between A's real-world objects (such as students) and C's computer bits, since *both* are semantic interpretations of the Pascal program.

What, then, is this relation D? It could be *simulation*: The computer bits simulate the student. But simulation is, after all, a kind of *implementation*. The computer bits are a computer implementation (an implementation in the medium of the computer) of the student.

2.2 Abstract Data Types.

The notion of an abstract data type (ADT) and its "implementation" is one of the most common uses of 'implementation'. There is a relatively informal use of the notion, as it appears in programming languages such as Pascal and as it is taught in introductory computer-science courses, and there is a more formal, mathematically precise use.

2.2.1 The informal notion of ADT implementation.

A stack is a particular kind of data structure, often thought of as consisting of a set of items structured like a stack of cafeteria trays: New items are added to the stack by "pushing" them on "top", and items can be removed from the stack only by "popping" them from the top. Thus, to define a stack, one needs (i) a way of referring to its top and (ii) operations for pushing new items onto the top and for popping items off the top. That, more or less (mostly less, since this is informal), is a stack defined as an ADT.

Now, Pascal does not have the stack as one of its built-in data types (as it does arrays, records, or sets). So, if you want to write a Pascal program that manipulates data structured as a stack, you need to "implement" a stack in Pascal. There are several ways to implement the ADT "Stack" in Pascal. Here's one way:

1. A Stack, s, is implemented as a 1-dimensional array, $A[0], \ldots, A[n]$, say, for some n;
2. $top(s)$ is defined to be a 1-argument function that takes as input the stack s and returns as output $A[0]$ (i.e., $A[0]$ is the implementation of the "top");
3. $push(s, i)$ is defined to be a 2-parameter procedure that takes as input the stack s and an item i (of the type allowed to be in the array), and yields as output the stack modified so that $A[0] := i$, and $A[j] := A[j-1]$ (i.e., each item on the stack is "pushed down");
4. and, almost finally, $pop(s)$ is defined to be a 1-argument function that takes as input a stack s and returns as output the item on the top of s (i.e., $top(s)$) while moving all the rest of the items "up" (i.e., $A[j] := A[j+1]$).

I said "almost finally" because—as should be obvious—some bookkeeping must be taken care of:

5. We have to specify what happens if the stack "overflows" (as when we try to push an $(n+2)$nd item onto a stack implemented as an $(n+1)$-element array).
6. We have to specify what happens to the "last" item when the top is popped (does the array cell that contained that item still contain it, or does it become empty?), etc.

These (as well as the limitations due to the type allowed to be in the array) are what are called "implementation details", since the ADT Stack "doesn't care" about them (i.e., doesn't—or doesn't *have to*—specify what to do in these cases).

Here's another way to implement a stack in Pascal. Do everything as before, but let $top(s) := A[n]$. This implementation of the Stack ADT is "inverted" with respect to the first one. The inversion, however, is (a) a ("mere") implementation detail and (b) undetectable in the program's input-output behavior. (This might remind you of inverted spectra.)

Here's a third way: Use Pascal's *pointer* data type to implement a stack as a "linked list". Here are a few of the details: First, a linked list ('list', for short) is itself an ADT. It is a sequence of items whose three basic operations are (1) *first*(l), which returns the first element on the list l, (2) $rest(l)$, which returns a list consisting of all the original items except the first, and (3) *make-list*(i, l) (or $cons(i, l)$), which recursively constructs a list by putting item i at the beginning of list l. Lists can be implemented in Pascal by, e.g., 2-dimensional arrays (here, the first item in each two-cell unit of the array is the list item itself, and the second item in the two-cell unit is an index to the location of the next item) or by means of "pointers" (each item on the list is implemented as a two-element "record", the first element of which is the list-item itself and the second element of which is a pointer to the next item). Finally, a stack s can be implemented as a list l, where $top(s) :=$ *first*(l), $push(s, i) :=$ *make-list*(l, i), and $pop(s)$ returns $top(s)$ and redefines the list to be $rest(l)$.

So: stacks can be implemented as arrays or as lists, and lists can be implemented as arrays or as records with pointers. ADTs can implement other ADTs, or they can be implemented "directly" in the given data structures of a programming language. What's going on here?

2.2.2 The formal notion of ADT implementation.

To see what's going on, we need to look at some of the more formal approaches to the definition and implementation of ADTs (a good survey is Morgado 1986).

Guttag et al. (1978: 61) assert that "the process of design (of data types) consists of specifying ... operations to increasingly greater levels of detail until an executable implementation is achieved." The implementation appears to be merely a more detailed version of the original "specification". The implementation details are essential for *executability* but not, presumably, for specifiability. So the implementation details serve a purpose, but one distinct from the original specification (or Abstraction).

What is the Abstraction? "A *data type specification* (or ADT) is a representation-independent formal definition of each operation of a data type. Thus, the complete design of a single data type would proceed by first giving its specification, followed by an (efficient) implementation that agrees with the specification" (Guttag et al. 1978: 61). So, implementations—the *detailed* specification of the operations—must "agree with" the *undetailed*—or *Abstract*—specification; the implementation must *satisfy* the definitions. That, of course, needs to be made precise, but it is more than suggestive of semantic interpretation. If the Abstraction is supposed to be "representation-independent", then perhaps an implementation *is* a representation. (There can be more than one implementation, e.g., "efficient" and inefficient ones.)

Guttag et al. (1978: 74) give an "example of the implementation of one data type, Queue ..., in terms of another, CircularLists." So, as noted before, ADTs can implement each other. This is done as follows: "We first give, in a notation very similar to that for the specification, an implementation of the Queue type consisting of a *representation* declaration and a *program* for each of the Queue operations in terms of the representation" (p. 74). In the example, the representation "medium" is CircularList, and the "programming language" consists of the operations of CircularLists. So, an implementation of an ADT consists of a representation and programs, where the programs implement the ADT's operations, as follows: Each operation of the ADT is ... defined? explicated? implemented? ... in terms of an operation of the implementing medium (the implementing ADT), *after* first representing each abstract-data-type entity (term) by a term of the implementing ADT. So, terms get interpreted by, or mapped into, elements of the interpreting domain, and predicates (operations) are mapped into predicates of the interpreting domain. On this view, implementation *is* semantic interpretation.

A more detailed and philosophically sophisticated approach is to be found in Goguen et al. 1978. The mathematical details are irrelevant to the present inquiry, but the overall picture they offer is useful, so let me attempt to summarize it here.

They begin by observing that "the term *abstraction* in computer science ... has been used in at least three ways which are distinct but related" (p. 82): An abstraction is either (1) "a mathematical model or description of something" (p. 82), or (2) "the process (or result) of generalizing" (p. 83), or (3) "a concept" considered "independent[ly] of its representation" (p. 83). Examples of (1) are "'abstract machines' as opposed to real hardware" (p. 82) and "abstract implementations", as "when one uses sets, sequences, or other mathematical entities to model some computational process or structure" (p. 83). So, a mathematical model is an abstraction of some real-world entity; as such, the abstraction seems to play the syntactic role. On the other hand, the implementation of a queue by a circular list (or a stack by a linked list) is an "abstract implementation", yet here it clearly plays the *semantic* role. (Cf. the "muddle of the model in the middle": Wartofsky 1979: xiii–xxvi, Rapaport 1995.)

In sense (2), abstractions ignore details. This contrasts nicely with the notion of an implementation as *providing* details. Presumably, an abstraction in sense (1) might ignore details, and hence be an abstraction in sense (2), but not necessarily. For instance, although the (admittedly informal) abstract notion of stack in §2.2 ignored the details imposed by the finiteness of the stack, we could have had an equally abstract presentation that payed attention to those details (yet could be implemented as a (finite) array, an "inverted" array, or a (finite) list). However, *merely* ignoring details does not by itself yield an abstract model in sense (1), because it might not be a *mathematical* model.

Goguen et al. take the third sense of 'abstraction' to be the relevant one for ADTs (cf. p. 81). The "representation" that such an abstraction is independent of has to do with notation, or the manner in which it is expressed:

> For example, "abstract syntax" considers syntactic structure independently of whether it is represented by derivation trees, parenthesized expressions, ... or whatever. This notion of abstract syntax is useful ... in specifying the semantics of a programming language in a manner independent of how it is implemented ....
>
> More to the point, an abstract data type is supposed to be independent of its representation, in the sense that details of how it is implemented are to be actually hidden or "shielded" from the user: He [sic] is provided with certain operations, and he only needs to know what they are supposed to do, not how they do it. (Goguen et al. 1978: 83; cf. Parnas 1972.)

That is, the programmer can deal directly with the ADT and ignore its implementation; one deals with it at a "high level". Consistent with our view that an implementation is a semantic interpretation, Goguen et al. observe "that what is usually called an 'abstract implementation,' that is, an implementation described by sets, sequences, etc., is *not* an 'abstraction' in the above sense; rather, it is a *particular*, but rather undetailed, implementation" (p. 83). So, an abstraction in sense (1) is not necessarily an abstraction in sense (3). It is undetailed, presumably because the implementing medium (the implementing ADT) is *itself* abstract (in sense (2)). Still, the mathematical model is a semantic model.

Now, what is this abstraction of the third kind? Goguen et al. note that it has to do with equivalence classes ("isomorphism classes", p. 83). They define an ADT as "the isomorphism class of an initial algebra in a category" of many-sorted algebras (pp. 88, 90). And they note that "An implementation is necessarily made within a specific framework, such as a particular programming language or machine" (p. 135); i.e., an implementation requires an implementing *medium*, or "framework".

Their mathematical "approach is to model an implementation framework as an algebra, with the elements of the carrier(s) being concrete data representations (machine states, primitive data types) and its operations the given basic operations (machine operations, basic instructions, programs) in these data representations" (p. 135). Note that they are *modeling* the implementing *medium* and that they do so by the *same* kind of entity as for an ADT, namely, an algebra! The implementation *itself*, of course, is something "physical"; it is merely being *described* algebraically.

The heart of the matter is expressed by them in their mathematical set-up as follows:

> Let B denote the implementation algebra .... [Let $T_{\Sigma,\epsilon}$ be] the specification algebra. The question now is, What relationship between $T_{\Sigma,\epsilon}$ and B constitutes an *implementation*? (p. 136.)

This is precisely the question: What is the relationship between an Abstraction (a "specification algebra") and an Implementation (an "implementation algebra")? (Note that B itself is (merely) a representation or model of the actual, physical implementations.) Goguen et al.'s answer is that the relationship is a structure consisting of B, a mapping from (roughly) $T_{\Sigma,\epsilon}$ to B, and a "congruence" (a family of equivalence relations on (roughly) T's image in B; p. 138). The core of this is, first, the mapping from the Abstraction to the Implementation, which is, on my theory and consistent with the view of Guttag et al., a *semantic interpretation*, and, second, the "congruence". The latter is a very special, intricate kind of isomorphism, one that "divides out" (they use quotient spaces) the "implementation details". So, B (or that which B is a mathematical model of) implements an Abstraction T if and only if B is a domain of semantic interpretation of T, ignoring the implementation details. Consider, e.g., the ADT Stack, and consider two specific implementations of it in Pascal, using an array $A[0], \ldots, A[n]$ with *top* implemented in one as $A[0]$ and in the other as $A[n]$. In both implementations, *top* is implemented as a specific element of the array. That it is $A[0]$ in one and $A[n]$ in the other is an implementation detail.

2.3 Implementation Outside of Computer Science.

2.3.1 Music.

Some of the clearest examples outside of computer science of what could be called 'implementation' come from music. This ought not to be surprising: After all, a music score is very much like a computer program or algorithm, and the musician-plus-instrument (or conductor-plus-orchestra) plays a role very much like that of the computer. A musical score is *not*, of course, *mathematically* an algorithm, since much is left open to "interpretation" by the musician (e.g, tempo, dynamics, optional repeats, phrasing, etc.). Nonetheless, it *is* a set of "instructions" that, when followed or executed, produce a certain output. The "process" consisting of the musician playing that music on an instrument can plausibly be said to *implement* the score. The score can be thought of (indeed, it *is*) a piece of syntax; the playing of the score provides a "semantic interpretation" of it.

An implementation requires an implementing medium. And, as should be evident, there can be many different media, hence many different implementations (the common core of which can be captured by the mathematical techniques of Goguen et al.). We find the same thing in music: A given score can normally be played on a variety of instruments, modulo a few changes necessitated by the nature of the instrument. Such changes, as well as the particular features of the instrument, constitute "implementation details". Often, these change the nature of the work, for good or bad: "a [piano] transcription [of a symphony] can hold a prism up to a familiar work, showing it in a new light" (Pincus 1990). That is, a piano transcription of a symphony is an *interpretation* of it—or, rather, *another* interpretation of "the work", i.e., of an ADT (the score) of which both the symphony *and* the piano transcription are (semantic) interpretations or implementations. The implementation is also, of course, an "interpretation" in the ordinary sense: Rosen (1991) speaks of "the essential gap between the composer's conception of a work of music and the *multiple possibilities of realizing it in sound*" (p. 50, my italics). The "conception" is the ADT; the "multiple possibilities" are different implementations.

Much the same can be said, *mutatis mutandis* for scripts and productions of plays (or scripts and movies). Where English talks of a *director*, French talks of a *réalisateur* (a realizer): At least for francophones, plays and movies are *implementations* (of scripts).

2.3.2 Language.

Language provides a variety of non-computer-science examples of implementation. For one thing, words can be considered as representations—hence, implementations—of ideas (cf. Harris 1987: xi, Ch. 6). For another, if language can be thought of as an Abstraction (as, perhaps, Chomsky's theory of universal grammar would have it), then it can be implemented in a variety of ways: first, by spoken languages (implemented in the medium of speech) as well as by signed languages (implemented in the medium of space; cf. Coughlin 1991), and, second, in many ways in both spoken and signed languages (e.g., French, English, etc., and American Sign Language, British Sign Language, etc.).

Sellars (1955 [1963] discussed another non–computer-science example of implementation:

> [In the context of chess,] ...attention must be called to the differences between 'bishop' and 'piece of wood of such and such shape'. ... [the former] belongs to the rule language of chess. And clearly the ability to respond to an object of a certain size and shape *as a bishop* presupposes the ability to respond to it as an object of that size and shape. But it should not be inferred that 'bishop' is 'shorthand' for 'wood of such and such size and shape' ... 'Bishop' is a counter in the rule language game and participates in linguistic moves in which ... the ... longer expression does not .... (Sellars 1955, §56 [1963: 343].)

"Being a bishop" is a nice example of what I am calling an Abstraction. Here, a bishop is implemented as a certain piece of wood. It could also, as Sellars observes, be implemented by a Pontiac if the chess game is played in Texas, where everything is supposed to be bigger:

> ... the term 'bishop' as it occurs in the language of both Texas [where it is "syntactically related ... to expressions mentioning different kinds of cars" (§59, p. 344)] and ordinary chess can be correctly said to have a common meaning—indeed to mean the bishop role, embodied in the one case by pieces of wood, and in the other by, say, Pontiacs .... (Sellars 1955, §62 [1963: 348].)

Here, we have an Abstraction (Chess) and two implementations (the ordinary Staunton pieces and the Texas pieces). We assume that the pieces that play the role of the bishop are both *called* 'bishops'; 'bishop' means the same thing in both implementations, namely, the Bishop Abstraction. That role is "embodied as"—i.e., *is implemented by*—a Pontiac in Texas and a certain ♗-shaped piece of wood in the Staunton set. The words 'bishop' as they occur in the two different languages refer to different entities (the language-entry and -departure rules in Sellars's language games differ).

3 POSSIBLE INTERPRETATIONS OF "IMPLEMENTATION".

Let's take another look at the *Oxford English Dictionary.* The *noun* 'implement' comes from the Latin for "a filling up", as in "that which serves to fill up or stock (a house, etc.)", and from the Old French for "to fill, fill up" in the sense of "completing" (Simpson & Weiner 1989, 7: 721). This suggests "filling in the details", which an implementation in the sense we are concerned with certainly does.

The *verb* is of more recent origin, having three senses, all with citations beginning in the 19th century (p. 722):

(a) "To complete, perform, carry into effect (a contract, agreement, etc.); to fulfil (an engagement or promise)." This is the earliest sense to be cited (1806)—implementing an obligation.

(b) "To carry out, execute (a piece of work)." Here, the citation is from 1837: implementing an invention.

(c) "To fulfil, satisfy (a condition)." This was used as early as 1857: implementing the "mechanical requisites of the barometer ... in ... an instrument".

Senses (b) and (c) seem closest to our concerns: Sense (b) relates to an Abstraction, and (c) relates to the implementation of an Abstraction—to satisfying the conditions of the Abstraction, or having the properties of the Abstraction. (More recent senses, with citations from 1926 and 1944, don't clarify much; curiously, none of the citations come from computer science.)

Recall that "implementation" is a relational notion, whose full context is always: I is an "implementation" of an "abstraction" A in some "medium" M. I have suggested that the notion of *Abstraction* be a generalization of the notion of an ADT. Must Abstractions be abstract, that is, non-spatiotemporal? If so, then they would contrast nicely with a *physical* or *concrete* interpretation of an "implementation"—i.e., with the "medium" always being spatiotemporal.

But we have seen that one Abstraction can implement another. So this characterization won't do. Instead, I suggest that we leave the notion unrefined for now except as that which can be implemented in some medium.

What about the medium? It could be abstract or concrete, giving rise to two varieties of implementation. An "abstract implementation" would be a specification, a filling-in of details, of an Abstraction. For instance, in top-down design, each level (except possibly for the last) is an "abstract implementation" of the previous one: I begin preparing my courses with a bare-bones course outline and successively refine it by adding details; or: I start solving a problem algorithmically by writing an algorithm in "pseudocode" and, by "stepwise refinement", fill in the details (e.g., pseudocode the procedures), until I finally encode it in, say, Lisp.

A "concrete implementation" would exist in a physical (or spatiotemporal) medium. It would necessarily have more details filled in, namely, those due to, i.e., contributed by, the medium. For example, my actually standing in front of the class, lecturing, is a concrete implementation of my final course outline. The actual words I say, the actual piece of chalk I use, etc., are all implementation details, filled in to "the last detail" by the very nature of the real, spatiotemporal events. Similarly, the actual execution of my Lisp program (perhaps after having been compiled into—i.e., further implemented in—ML)—the *process*—is its concrete implementation. Both abstract and concrete implementations are semantic interpretations.

Is "implementation" a concept *sui generis*? Or should it be assimilated to some other, perhaps more familiar, notion, such as "instance", "exemplification", "reduction", "supervenience", etc.? There is very little agreement over the proper characterization of those other, candidate notions, or even over terminology. For instance, it seems clear that an implementation of an Abstraction is *not* an "instance" or "instantiation" of the Abstraction, because two Abstractions (e.g., two ADTs) can implement *each other*. As we saw, the ADT Record can be used to implement the ADT List. Moreover, the ADT List can be used to implement the ADT Record. And, though there is probably no good reason to do so, one could, perversely, implement lists by records that are themselves implemented by lists. And so on. Yet "instantiation" is normally thought of as an asymmetric relation. In spite of this, we find recognized authorities on implementations, Guttag et al., saying that an implementation *is* an "instance" (Guttag et al. 1978: 62). Let's explore these issues.

3.1 Implementation as Individuation.

Let's begin by distinguishing between "individuation" and "differentiation" (cf. Castañeda 1975). In Porphyry's Tree, that early ancestor of semantic networks, a universal, such as a genus, is analyzed into sub-genera or species by means of a "specific difference" or *differentia*. Thus, for example, the *differentia* Rational applied to the genus Animal yields the species Human (= Rational Animal); all other, non-human, animals are *not* Rational. Thus, Humans are *differentiated* from non-Humans. As a category, Human is "lower" than Animal; it is more "specific"—it has an extra defining property, namely, being rational. Human is itself a universal—as it happens, an *infimum species*, i.e., a category that is not analyzed into subcategories but into concrete *individuals*, e.g., Plato, Sappho, you, me.

What is the analogue of a *differentia* that, when applied to an *infimum species* yields an individual? Duns Scotus called it 'haecceity', or "thisness". "Instantiation" is the relation between any level of Porphyry's Tree and the level *below* it; "differentiation" is a relation between subcategories (or members) of a single category. Thus, just as Human is differentiated from non-Rational Animal, so Plato is differentiated from Sappho, and you from me. And just as Human is instantiated from (or, is an instance of) Animal, so Plato, Sappho, you, and I are instantiated from (or, are instances of) Human. And Plato et al., unlike Human et al., are "individuals": "Individuation" is the relation between an *infimum species* and its individuals.

Thus, perhaps, implementations are individuals, and Abstractions are universals. That does seem to hold for *concrete* implementations. But it fails to hold for *abstract* implementations, and it only works when there is a hierarchy or linear ordering of successively more detailed Abstractions. It fails to account for the relation that obtains when a list implements a stack.

On the other hand, since individuals and lower-level instantiables *can* be viewed as *implementations* of higher-level instantiables or universals, I suggest that instances and individuals are implementations, but not conversely.

3.2 Implementation as Instantiation.

Anthony 1991 explicitly argues that computer "implementations" are *not* "instantiations". The background of Anthony's argument is whether "a Connectionist architecture *instantiates* the Classical framework" or whether "a Connectionist architecture *implements* a Classical architecture" (p. 325, my italics), or whether there is some other (or no) relation between them.

As Anthony uses the term,

> 'Instantiation' expresses a simple relation between individuals and properties: an individual i instantiates a property P if and only if Pi. ... In the case of instantiation ... a *single* model or architecture is involved, and what is in question are its properties. (p. 325.)

In §3.1, instantiation was a relation between an individual or a category and its immediate superordinate category. Here, it is a relation between an individual and a *property*. Others have called the latter relation 'exemplification', though, of course, the relation between an individual and its properties can be (and has historically been) explicated in terms of category membership, and "exemplification" is the term often used for the relation between a real object and the Platonic Form that it "participates" in. Rather than trying to resolve several thousand years of metaphysics, let's stick with Anthony's definition for now.

So, for a connectionist architecture to instantiate a classical framework would be for it to have classical properties. Let P_{cl} be the set of properties that "define the Classical ... framework" (p. 325). Let C_x be "a particular Connectionist architecture". Suppose that C_x has all of P_{cl}. Couldn't we then identify the Abstraction ClassicalSystem as the set of properties P_{cl} and treat C_x as an implementation of it? Anthony observes in a footnote (p. 339n7) that "individuals" could be "*abstract* objects like functional architectures" (my italics); thus, C_x is *also* an Abstraction (thus, clearly, Anthony is not speaking the language of §3.1).

In contrast,

> Where *implementation* is at issue ..., *two* functional architectures must be considered.

> A functional architecture FA1 is implemented, if at all, by the execution of a program in a distinct functional architecture FA2. (p. 325.)

So, FA2 might *itself* not have FA1's properties (so FA2 need not be an *instance* of FA1), but the *process*—the program in execution—*might* have FA1's properties (and so be an instance of FA1). In general, this seems OK. For instance, an ML program might have FA1's properties, but the ML *itself* might not. As a trivial example, an ML program can have records, while the ML itself doesn't have them.

"Intuitively," Anthony tells us, "the primitive operations, representational structures, etc. of FA1 get 'made up' or 'constructed' out of the resources of FA2. ... This is the relation that typically exists, for example, between assembly language functional architectures ... and higher-level architectures like LISP or Pascal ... when the latter are up and running on a computer" (p. 325). So the idea is this: If FA2 (e.g., the machine language) has records as a primitive data type, then it's easy to implement FA1 (e.g, Pascal) in it, because they both already share the same properties—they both instantiate "having the record data type". If FA2 *lacks* records, they can nonetheless be implemented in it. But wouldn't FA2 then *have* records? Anthony seems to be trying to distinguish between essential properties and accidental ones: Records are an "essential" feature of a programming language that has them among its primitive (or built-in) data types, an "accidental" feature of a programming language that can only define them in terms of (or construct—or implement!—them out of) its primitive ones.

Consider, by way of analogy, the rationals and the integers. As is well known, the rationals can be ... implemented in? constructed out of? (cf. §3.3) ... (equivalence classes of ordered pairs of) integers (and vice versa, don't forget!—since $\mathbb{Z} \subset \mathbb{Q}$). Suppose we have a *language* for talking about the rationals (and sets). It will have, as one of its terms, a representation for $\frac{1}{2}$ (i.e., the rationals have, as one of its data objects, $\frac{1}{2}$). Suppose that the language also has as a primitive predicate some expression for a property P that applies only to rationals (i.e., the rationals have, as one of their properties, P)—e.g., (some) rationals have the property of being less than 1, and the set of rationals itself has the property of being dense. Now, suppose we have a language for talking about the integers (and sets). Can we talk about the *rationals* in the language of integers? Yes—by finding or constructing (analyzing?) the rationals' data objects (like $\frac{1}{2}$) in the integers and by finding or constructing (or analyzing) the rationals' properties among (or from) the integers' properties. By defining new *terms* in the language of integers, that language would now have terms for $\frac{1}{2}$ and P. That is, we could now say more in the language of integers than we thought we could; it *wasn't*, after all, limited to talking about integers. What, then, would be the difference between the language of rationals and the (extended) language of integers? The former would have certain terms and predicates (NPs and VPs) that the latter would lack; but they could be *defined* in the language of integers.

That this really is close to what Anthony has in mind can be seen from the following passage:

> ...in cases of *implementation*, lower-level architectures typically do not *instantiate* the *characteristic* properties of higher-level ones. An assembly-level architecture *implementing* LISP, for instance, does not also *instantiate* LISP: it lacks the *necessary* primitive properties (e.g., CAR, CDR), and has primitive operations LISP lacks (e.g., various operations on the contents of the accumulator). (p. 326, my italics.)

Here, 'characteristic' and 'necessary' can be taken to mean "essential". But doesn't an ML implementation of Lisp *have* the Lisp function `car`?[5] Maybe not: It can "simulate" `car`—or implement it?—but it doesn't *have* it; it can do what `car` does without *having* `car`. If you'll excuse the pun, I can do what can be done with a car without having one—by walking, taking the bus, etc.

We can draw a distinction between "weak" and "strong" implementations. For instance, a strong implementation of Lisp in ML would be such that the ML actually *had* identifiable data structures and procedures corresponding to lists, `car`, etc. A weak implementation of Lisp in ML would be such that it would do the same things (e.g., be able to return the first element of a list) without having lists or `car` (just as I can get from my home to a store by car or by walking).

Conversely, "it is also true that an instantiation of LISP need not implement any distinct, higher-level LISP architecture" (p. 326). For example, I suppose, Allegro Common Lisp (ACL) (understood as an *instantiation* (rather than an *implementation*?) of Lisp) need not *implement* SNePS (a semantic-network knowledge-representation and reasoning system written in ACL; cf. Shapiro 1979; Shapiro & Rapaport 1987, 1992). So, instances and implementations (as Anthony defines them) "are mutually independent" (p. 326).

Let's return to the rationals and the integers. Consider the integers first (and, for convenience, consider only the non-negative integers). What are they? One way to answer this is to cite Peano's axioms. That would be to present the Abstraction Integers—in fact, it is an ADT. Another way to answer the question is to say that integers are any things that satisfy Peano's axioms. So, e.g., the sequence consisting of $\emptyset$, $\{\emptyset\}, \{\{\emptyset\}\}$, etc., are integers. So is the sequence $\emptyset, \{\emptyset\}, \{\emptyset, \{\emptyset\}\}$, etc. So is the sequence $\emptyset, \{x \mid x \text{ is a set } \& \textit{ Cardinality}(x) = 1\}$, $\{x \mid x \text{ is a set } \& \textit{ Cardinality}(x) = 2\}$, etc. So is the sequence of symbol types 0, 1, 2, ..., 10, 11, 12, ..., 99, 100, etc. So is the sequence of symbol types 0, 1, 10, 11, 100, etc. (And if you ignore 0, so is I, II, III, ..., X, etc.). And so on. The *differences* between each of these are "implementation details".

The vague feeling of discontent that this leaves us with is that there are "*too many*" integers; But model-theoretic semantics teaches us that there isn't any "intended interpretation". For any set of axioms, there are infinitely many models, including non-isomorphic ones. So, the only way to talk about "the" integers is to restrict ourselves to talk about Peano's axioms. The alternative is to choose, arbitrarily, some model of them and talk about *it*.

Now, since such axioms are ADTs, such models are *implementations* in various media. For our first three examples above, the implementing media are sets "put together" in different ways; for the others, the implementing media are certain symbol types. If we restricted ourselves to some finite initial sub-sequence of the non-negative integers, we could take arbitrary physical objects as our implementing medium.

Are any of these implementations "instances" of ... integers? Of Peano's axioms? 'Instance' in the Porphyrian-tree sense? In Anthony's sense? I'm inclined to say 'No', but that's primarily because I find the interpretation of 'implementation' in terms of semantic models to be more illuminating. Moreover, I suspect that if one wanted to force the concept of an implementation into the mold of "instantiations", one could do so only by seeing "instantiations" as a kind of semantic modeling.

3.3 Implementation as Reduction.

Now consider the (non-negative) rationals. Consider a set of axioms for the rationals; i.e., consider the ADT (NonNegative)Rationals. What are some of its implementations? Well, there are the fractions, i.e., the symbol types $\frac{0}{4}, \frac{1}{2}, \frac{2}{3}, \frac{1}{7}$, etc. There are the repeating decimals, i.e., the symbol types $0.\overline{0}$, $0.5\overline{0}$, $0.\overline{6}$, $0.\overline{142857}$, etc. There are also certain *constructions* from the integers, e.g., certain ordered pairs of integers: $\langle 0,\ 4\rangle$, $\langle 1,\ 2\rangle$, $\langle 2,\ 3\rangle$ $\langle 1, 7\rangle$, etc. Each of these can be considered to be an *implementation* of the rationals. Rational numbers are anything that satisfy the axioms.

Here, the notion of implementation details plays a larger role, since we seem to have "too many" rationals. In a Morning Star/Evening Star sense, $\frac{2}{3}$ and $\frac{4}{6}$ are the "same" rational number, as are the ordered pairs $\langle 2,\ 3\rangle$ and $\langle 4,\ 6\rangle$. We could say that that's an implementation detail, and provide rules (further axioms?) to indicate when two "intensionally distinct" rationals are "extensionally equivalent". We have such rules for, say, addition of integers: Does '$2+2=4$' state a fact about addition, or does it assert an extensional equivalence between intensionally distinct integers? Or else we could—as in fact we normally do—implement the rationals as *equivalence classes* of ordered pairs of integers.

Now, often this "implementation" of rationals by integers (plus set theory), is called a "reduction" of the rationals to the integers. "All we really need," so the reductionist says, "are the integers (and set theory); we can define the rationals in terms of them (or, we can reduce the rationals to them)." So: Is implementation just reduction? Are all reductions implementations?

Again, we have related, but distinct, concepts. Smith notes that "*Reducibility* ... is a relation between *theories*; one theory is reducible to another if, very roughly, its predicates and claims can be translated into those of another" (Smith 1991: 280n39). Now, in the case of the rationals and the integers, I would really hesitate to say that the former have been "reduced" to the latter. I would be willing to say that the *theory* of rationals can be reduced to the *theory* of integers-plus-sets. But even here, when we prove some theorem about rationals, we haven't proved a theorem about integers but, at best, about certain *sets* whose "ground elements" are integers. For example, to prove a theorem about the *rational* number $\frac{1}{2}$ would be to prove a theorem about the following arcane set of sets of integers and sets of integers: $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^{+}\ \&\ 2a = b\}$.[6] Suppose integers are implemented as sets, and multiplication is implemented as a set of ordered pairs of factors. Then we might have the following situation: If $\{\{\}\}$ and $\{\{\}, \{\{\}\}\}$ implement 1 and 2, respectively, then $\frac{1}{2}$ could be implemented as the monstrosity $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^{+}\ \&\ \{\{\{\}, \{\{\}\}\}, \{\{\{\}, \{\{\}\}\}, a\}\} = b\}$. The mind boggles. Is this supposed to be *easier* to understand than '$\frac{1}{2}$' or '$0.5\overline{0}$'?

And the only reason we're interested in those rather arcane sets "of" integers is because they implement—are models of—the ADT Rationals. We might feel more "comfortable" with these arcane sets insofar as we are more comfortable with good old-fashioned sets and integers rather than with rationals *per se*. But that is an epistemological consideration that is rather suspect in the long run.

Once we have implemented the rationals using integers and sets, we also have another implementation of the integers, of course (since the integers are a proper subset of the rationals—or perhaps it would be better to say that a certain proper subet of the rationals is an implementation of the Integer ADT). I have in mind here the sequence $\frac{0}{1}, \frac{1}{1}, \frac{2}{1}, \frac{3}{1}$, etc. As a matter of fact, there are several implementations of the integers to be found among the rationals; here are a few:

$$\frac{0}{2}, \frac{1}{2}, \frac{2}{2}, \frac{3}{2}, \ldots$$

$$\frac{0}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{3}, \ldots$$

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \ldots$$

These (especially the last one) may seem a bit odd, but recall that Peano's axioms only require that there be a successor relation, not that that relation be (implemented as) +1 (or even as +Successor(0)); the choice of a 0-element is arbitrary, too. Other arbitrarily strange ones are possible, e.g.,

$$\frac{0}{1}, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \ldots$$

And, of course, using an order imposed by a diagonalization, the (non-negative) rationals themselves can be taken as an implementation of the integers; e.g., if we arrange them (ignoring equivalences) two-dimensionally as follows:

$$\frac{0}{1}, \frac{0}{2}, \frac{0}{3}, \frac{0}{4}, \ldots$$

$$\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots$$

$$\frac{2}{1}, \frac{2}{2}, \frac{2}{3}, \frac{2}{4}, \ldots$$

then the sequence $\frac{0}{1}, \frac{0}{2}, \frac{1}{1}, \frac{0}{3}, \frac{1}{2}, \frac{2}{1}, \frac{0}{4}, \frac{1}{3}, \frac{2}{2}, \ldots$ implements the integers, too.

Finally, we could, if we wanted to, *re-implement* the rationals in one of these implementations of the integers, by the usual ordered-pair construction.

Why bother? Well, besides whatever insights such playful model-making gives us into the logical structure of the integers, it also shows that *reduction* (or construction) for the purposes of providing stronger epistemological foundations is *not* what implementation is. All of the above are implementations; none serves any interesting or useful reductive purposes.

The upshot is that although some, or even all, reductions or constructions might be implementations, certainly not all implementations are reductions.

3.4 Implementation as Supervenience.

Recall that an implementation of an Abstraction in some medium is a semantic model of the Abstraction in the "medium" of some semantic domain. And a semantic model is *any* structure—including the Abstraction itself!—that can be correlated (or put into correspondence) with the Abstraction. The closer the correlation, the better the semantic interpretation, even if, in the base case of a *self*-interpretation, we must resort to syntactic understanding (i.e., understanding via familiarity with the syntactically legal symbol manipulations; cf. Rapaport 1986, 1988, 1995). One major correlation relation that is a plausible candidate for interpreting implementation is supervenience. As Smith notes, "the term *supervenience* is used to relate phenomena themselves; thus the strength of a beam would be said to supervene on the chemical bonds in the constitutive wood. ... [S]upervenience doesn't necessarily imply reducibility" (Smith 1991: 280n39); neither does implementation (§3.3). When one domain "supervenes" on another, is it *implemented* by that other domain? And when one domain is *implemented* by another, does it supervene on that other domain? What, then, is supervenience?

3.4.1 Supervenience: An introduction.

Kim 1978 gives a precise formulation of the informal notion that "one *family of properties* is 'supervenient' upon another *family of properties* in the sense that two things alike with respect to the second must be alike with respect to the first" (p. 149, my italics), even though "there is no relationship of definability or entailment between the two families" of properties (pp. 149–150). Now, implementation is neither definability nor entailment, so it could indeed be supervenience. According to Kim, "the main point of the talk of supervenience is to have a relationship of dependence or determination between two families of properties *without* property-to-property connections [or "correlations"] between the families" (p. 150). But in the case of implementation there *are* such property-to-property correlations. So maybe implementation *isn't* supervenience? As we will see, however, Kim's explication of supervenience allows for such correlations.

Kim first defines two set-operations, $^{\#}$ and * (Kim 1978: 152, col. 1): Where M is a set of properties, $M^{\#}$ is its "closure ... under the usual Boolean operations", and M^{*} ($\subseteq M^{\#}$) is the set of "M-maximal properties"; i.e., "if M is finite, each member of M^{*} is a maximal consistent conjunction of the properties, and the complements of the properties, in M; if M is not finite, the members of M^{*} are maximal consistent sets of the properties in M and their complements". Consider an example. Let P, Q be properties, and let $M = \{P,\ Q\}$. Then $M^{\#} = \{P,\ Q,\ P \wedge Q,\ P \vee Q,\ P \rightarrow Q,\ \neg P,\ \neg Q,\ \neg(P \wedge Q),\ \neg(P \vee Q),\ \ldots\}$, and $M^{*} = \{P \wedge Q \wedge (P \vee Q) \wedge \ldots,\ (P \wedge \neg Q) \wedge (P \vee \neg Q) \wedge \ldots,\ \ldots\}$, where each element of M^{*} is an M-maximal property (and—in our example—the first-listed element of M^{*} contains no occurrences of $\neg P$ or $\neg Q$, and the second-listed contains no occurrences of $\neg P$).

Next, let D be a domain of objects, and let M, N be sets of properties that elements of D can have. Then *M is supervenient on N with respect to D* $=_{df}$ $\Box$(objects in D that share all properties in $N^{\#}$ also share all properties in $M^{\#}$) (Kim 1978: 152, col. 1). That is, suppose M supervenes on N with respect to D, and let $d, d' \in D$. Then $\Box$(d, d' share all properties in $N^{\#} \rightarrow d, d'$ share all properties in $M^{\#}$).[7] What's meant is not that d, d' *have* all properties in $N^{\#}$, but that *if* they have all and only the *same* properties in $N^{\#}$, then they also have the same properties in $M^{\#}$. So, where D, d, d', N, M are as before, *M supervenes on N with respect to D* $=_{df}$ $\Box((\forall \mathrm{P}_N \in N^{\#})[\mathrm{P}_N(d) \leftrightarrow \mathrm{P}_N(d')] \rightarrow (\forall \mathrm{P}_M \in M^{\#})[\mathrm{P}_M(d) \leftrightarrow \mathrm{P}_M(d')])$.

Kim presents an argument that reducibility and definability entail supervenience (Kim 1978: 152, col. 1). Can we run a similar argument to show that if M is *implemented* by N, then M supervenes on N? The argument requires biconditionals between N and M. Surely, if N implements M, such biconditionals would be provided for by the semantic interpretation function between N and M. Suppose that two things diverge on some M-property. Then they'll diverge in $N^{\#}$. So, if there are such biconditionals, then implementation does entail supervenience. Are there really such biconditionals? Since N implements M, there could be implementation side-effects (the domain of semantic interpretation might be "bigger" than the image of M in it). Still, if things diverge on M, they'll diverge in $N^{\#}$ (though perhaps not conversely).

Kim argues that supervenience on a *finite* N entails that "each property in M which is instantiated is biconditional-correlated with some property in $N^{\#}$" and that such generalizations are lawlike (p. 152, col. 2). This is surely true for implementation in the N-to-M direction (p. 152, col. 1). Is it true in the M-to-N direction (p. 152, col. 2)? Suppose that $Q_1, \ldots, Q_n$ are the physical properties of the implementation, that P is an M-property, and that $Q_1 \vee \ldots \vee Q_n \rightarrow P$. Suppose, by way of contradiction, that x (e.g., a computer process) has P (e.g., a certain input-output behavior) but that x lacks each Q_i (i.e., is implemented differently). However, x is implemented *somehow*; let K be a property that x has in virtue of its implementation. Suppose y (some other process) also has K. Now, since M supervenes on N (N implements M), y has P (i.e., y has x's input-output behavior). So, K must be one of the Q_is. Thus, Kim's argument seems to carry over (although details of the relationships between M, N and P, Q are not clear).

Moreover, supervenience *is* a semantic relation:

> To summarize: (1) if M supervenes on N, there are property-to-property correlations between M and N; (2) every property in M has either a necessary or sufficient condition in N ...; (3) if N is finite, every property in M is biconditional-connected with some property in N. ... [F]inite-based supervenience ... guarantees for each property in the supervenient family a co-extension in the supervenience base; and depending on the modality that attaches to the correlations between the two sets of properties, this may yield reducibility and definability. (Kim 1978: 153–154.)

Kim introduced supervenience to explain the relationship of mind to body. Viewing the supervenient set as the mental realm and the supervenience base as the physical realm, each mental property has a co-extensive physical property and might be reducible to it, or definable in terms of it. The co-extensiveness *almost* works for implementation, but, strictly speaking, it doesn't. For the implementing device is not a set of properties; hence, it has no extension. Rather, it *is* the extension of the mental (or Abstract) properties. It is an open question what the appropriate "modality" is for sets of mental properties and sets of physical properties.

But there is a problem in assimilating implementation to supervenience: Implementation isn't a relation between sets of properties. It's a relation between "physical" things and Abstractions—a relation between two different *kinds* of things—whereas supervenience is a relation between sets of properties. Let A be an Abstraction and I be its implementation in some medium S. Then it is S that has both A-properties and I-properties. So A-properties *could* supervene on I-properties. So, possibly, A is implemented by I if and only if A-properties supervene on I-properties. Indeed, Kim (1979: 43–44) sees supervenience as a very general version of the family of concepts that includes reducibility, etc. So perhaps it is the base relation in terms of which the others can be defined? I am uncomfortable with this, primarily because I see the generalized *semantic* relation as the fundamental one, and I take implementation to be a specific case of a semantic relation. So, too, for supervenience, which is, as we saw, a correlation relation.

A problem with supervenience as defined above is that there can be two "physically indistinguishable worlds" that are not also "psychologically indistinguishable" (Kim 1979: 40). Kim offers 'strong supervenience' as a remedy:

> A *strongly supervenes* on B just in case necessarily for each x and each property F in A, if x has F, then there exists a property G in B such that x has G, and *necessarily* if any y has G it has F. (Kim 1983: 49.)

and he points out that "Both relations are transitive, reflexive, but neither symmetric nor asymmetric" (p. 49). Transitivity is good; it's needed to account for levels of virtual machines, each of which can be said to supervene on, or be implemented by, a lower-level machine. Reflexivity, though, does not seem to be a property that we would want implementation to have. This means that *every* implementation *must* have implementation-dependent side-effects, since every implementation of an Abstraction will contribute something over and above what the Abstraction specifies.

If supervenience is non-symmetric, then it's possible for two properties to supervene on each other. Could each be an implementation of the other? That *seems* counterintuitive. Surely, two things can implement each other—or be semantic interpretations of each other—but not at the same time. There is a directionality, a point of view of the third party that *uses* one domain as a semantic interpretation or implementation of the other—recall the discussions of the asymmetry of antecedent understanding (cf. Rapaport 1995). So it looks as if supervenience is *not* implementation.

3.5 Summary.

The implementation relation is a widespread phenomenon, taking many guises. It is a relation that obtains between two things—I have called them the Abstraction and the Implementation—when the Implementation is a "concrete" or "real" or "physical" thing that has all the properties of the Abstraction. But we have also seen that Implementations can be equally "abstract". So there are two sorts of implementations: abstract and concrete ones, the latter being "realizations" in some physical medium. We have seen that they typically have *more* properties than their Abstraction. So perhaps the implementation relation is best construed (even etymologically) as a general term for *any* filling in of details; concrete implementations are fillings-in in concrete media. Thus, the notion of implementation comes along with a notion of "level": the more detailed level being "below" the "higher" (or more abstract)level, and the "concrete" or "physical" level being at the "bottom"—being the "foundation" as it were.

We have also seen that individuation, instantiation, reduction, and supervenience are all related to implementation, though "weaker" than it. The single best "interpretation" of implementation seems to be that of semantic interpretation: I is an implementation of A in medium M if and only if I is a semantic interpretation or model of A, where A is some syntactic domain and M is the semantic domain.[8]

NOTES

[1]This can be generalized to different possible worlds. Thus, intelligence, say, could be realized in several different (physical) media: This is the notion of "multiple realizability".

[2]I am using 'ML' to abbreviate 'machine language', not to refer to the programming language named "ML".

[3]John Sowa, personal communication, 29 November 1993.

[4]Due to my colleague Bharadwaj Jayaraman (personal communication).

[5]The Lisp function `car` (or `first`) takes a list as input and returns its first member; the Lisp

function `cdr` (or `rest`) takes a list as input and returns the "rest of" that list, i.e., the list consisting of all but that first member.

[6]The ordered pair $\langle 1,\ 2\rangle$ "is" (or can be implemented as!) $\{1, \{1, 2\}\}$. The equivalence class containing $\langle 1,\ 2\rangle$ "is" $\{\langle a, b\rangle \mid a, b \in \mathbb{Z}^+ \ \&\ 2a = b\}$. So, the rational $\frac{1}{2}$ "is" $\{\{a, \{a, b\}\} \mid a, b \in \mathbb{Z}^+ \ \&\ 2a = b\}$.

[7]That, at least, is what Kim *says*; but doesn't he mean $M*$ and $N*$? Perhaps not; cf. Kim 1978: 153, col. 1.

[8]In a sequel to this essay, I examine some of the implications of this point of view: the role of the "implementation details", the question of whether an implementation is "the real thing", and the problem of whether anything can be an implementation of anything else (Rapaport, in preparation, Ch. 7). I am grateful to my colleague Stuart C. Shapiro for comments on an earlier version of this essay.

References

1. Anthony, Michael V. (1991), "Fodor and Pylyshyn on Connectionism," *Minds and Machines* 1: 321–341.

2. Castañeda, Hector-Neri (1975), "Individuals and Non-Identity: A New Look," *American Philosophical Quarterly* 12: 131–140.

3. Coughlin, Ellen K. (1991), "A Professor Champions Distinct Culture of Deaf People," *Chronicle of Higher Education* (2 October 1991): A5.

4. Goguen, J. A.; Thatcher, J. W.; & Wagner, E. G. (1978), "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in Raymond T. Yeh (ed.), *Current Trends in Programming Methodology*, Vol. IV: *Data Structuring* (Englewood Cliffs, NJ: Prentice-Hall): 80–149.

5. Guttag, John V.; Horowitz, Ellis; & Musser, David R. (1978), "The Design of Data Type Specifications," in Raymond T. Yeh (ed.), *Current Trends in Programming Methodology*, Vol. IV: *Data Structuring* (Englewood Cliffs, NJ: Prentice-Hall): 60–79.

6. Harris, Roy (1987), *Reading Saussure: A Critical Commentary on the* Cours de Linguistique Générale (La Salle, IL: Open Court).

7. Hayes, John P. (1988), *Computer Architecture and Organization, 2nd edition* (New York: McGraw-Hill).

8. Jennings, Richard C. (1985), "Translation, Interpretation and Understanding," paper read at the American Philosophical Association Eastern Division (Washington, DC); abstract, *Proceedings and Addresses of the American Philosophical Association* 59 (1985) 345–346.

9. Kim, Jaegwon (1978), "Supervienience and Nomological Incommensurables," *American Philosophical Quarterly* 15: 149–156.

10. Kim, Jaegwon (1979), "Causality, Identity, and Supervenience in the Mind–Body Problem," in Peter A. French, Theodore E. Uehling, Jr., & Howard K. Wettstein (eds.), *Studies in*

Metaphysics, Midwest Studies in Philosophy, Vol. 4 (Minneapolis: University of Minnesota Press): 31–49.

11. Kim, Jaegwon (1983), "Supervenience and Supervenient Causation," *Southern Journal of Philosophy* 22, Supplement, pp. 45–56.

12. Marcotty, Michael, & Ledgard, Henry (1986), *Programming Landscape: Syntax, Semantics, and Implementation, 2nd edition* (Chicago: Science Research Associates).

13. Morgado, Ernesto J. M. (1986), "Semantic Networks as Abstract Data Types", *Technical Report* 86-19 (Buffalo: SUNY Buffalo Department of Computer Science).

14. Parnas, David (1972), "A Technique for Software Module Specification with Examples," *Communications of the Association for Computing Machinery* 15: 330–336.

15. Pincus, Andrew L. (1990), "The Art of Transcription Sheds New Light on Old Work," *The New York Times*, Arts and Leisure (Sect. 2) (23 September 1990).

16. Rapaport, William J. (1986), "Searle's Experiments with Thought," *Philosophy of Science* 53: 271–279.

17. Rapaport, William J. (1988), "Syntactic Semantics: Foundations of Computational Natural-Language Understanding," in James H. Fetzer (ed.), *Aspects of Artificial Intelligence* (Dordrecht, Holland: Kluwer Academic Publishers): 81–131; reprinted in Eric Dietrich (ed.), *Thinking Computers and Virtual Persons: Essays on the Intentionality of Machines* (San Diego: Academic Press, 1994): 225–273.

18. Rapaport, William J. (1995), "Understanding Understanding: Syntactic Semantics and Computational Cognition", in James E. Tomberlin (ed.), *AI, Connectionism, and Philosophical Psychology*, Philosophical Perspectives, Vol. 9 (Atascadero, CA: Ridgeview): 49–88; to be reprinted in Andy Clark & Josefa Toribio (1998), *Language and Meaning in Cognitive Science: Cognitive Issues and Semantic Theory*, Artificial Intelligence and Cognitive Science: Conceptual Issues, Vol. 4 (Hamden, CT: Garland).

19. Rapaport, William J. (forthcoming), "How Minds Can Be Computational Systems", *Journal of Experimental and Theoretical Artificial Intelligence*.

20. Rapaport, William J. (in preparation), *Understanding Understanding: Semantics, Computation, and Cognition*; unpublished ms. available from the author.

21. Rosen, Charles (1991), Reply to letter, *New York Review of Books* (14 February 1991): 50.

22. Sellars, Wilfrid (1955), "Some Reflections on Language Games," in *Science, Perception and Reality* (London: Routledge & Kegan Paul, 1963): 321–358.

23. Shapiro, Stuart C. (1979), "The SNePS Semantic Network Processing System," in Nicholas Findler (ed.), *Associative Networks: Representation and Use of Knowledge by Computers* (New York: Academic Press): 179–203.

24. Shapiro, Stuart C., & Rapaport, William J. (1987), "SNePS Considered as a Fully Intensional Propositional Semantic Network," in Nick Cercone & Gordon McCalla (eds.), *The Knowledge*

Frontier: Essays in the Representation of Knowledge (New York: Springer-Verlag): 262–315; shorter version appeared in *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86, Philadelphia)* (Los Altos, CA: Morgan Kaufmann): 278–283; revised version of the shorter version appears as "A Fully Intensional Propositional Semantic Network," in Leslie Burkholder (ed.), *Philosophy and the Computer* (Boulder, CO: Westview Press, 1992): 75–91.

25. Shapiro, Stuart C., & Rapaport, William J. (1992), "The SNePS Family," *Computers and Mathematics with Applications* 23: 243–275; reprinted in Fritz Lehmann (ed.), *Semantic Networks in Artificial Intelligence* (Oxford: Pergamon Press, 1992): 243–275.

26. Simpson, J. A., & Weiner, E. S. C. (preparers) (1989), *The Oxford English Dictionary, 2nd edition* (Oxford: Clarendon Press).

27. Smith, Brian Cantwell (1987), "The Correspondence Continuum," *Report CSLI-87-71* (Stanford, CA: Center for the Study of Language and Information).

28. Smith, Brian Cantwell (1991), "The Owl and the Electric Encyclopedia," *Artificial Intelligence* 47: 251–288.

29. Smith, Brian Cantwell (1997), "One Hundred Billion Lines of C++", *CogSci News* (Bethlehem, PA: Lehigh University Cognitive Science Program), Vol. 10, No. 1 (Spring): 2–6.

30. Soare, Robert I. (1996), "Computability and Recursion," *Bulletin of Symbolic Logic* 2: 284–321.

31. Wartofsky, Marx W. (1979), *Models: Representation and the Scientific Understanding* (Dordrecht, Holland: D. Reidel, 1979).