IMPLEMENTATION IS SEMANTIC INTERPRETATION

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

1. Introduction

Consider the implementation relationships among algorithms, computer programs, and the computers that execute them. An algorithm is (roughly) a procedure for computing a function.[1] A program that is said to "implement" the algorithm in some programming language is a more detailed *textual* expression of the algorithm. A computer *process*[2]—an algorithm being executed—is a physical device (a computer) *behaving* in a certain *way* that is specified by the program; the physical device running the process "implements" the program.

Here is another example from computer science: 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. The following non-computer-science examples are clearly of the same type as these paradigms, even though we don't, normally, use the term 'implementation' in discussing them: A performance is an "implementation" of a musical score or play-script. A house is an "implementation" of a blueprint. A set-theoretic model is an "implementation" of a formal theory.[3]

But what is "implementation"? Is it *sui generis*? Is it individuation? Instantiation? Reduction? Supervenience? 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");[4] a *semantic domain*, similarly characterized; and a

"Implementation is Semantic Interpretation" by William J. Rapaport,
The Monist, vol. 82, no. 1, pp. 109–130.

relation of semantic interpretation that maps the former to the latter.[5] Put this way, there is no *intrinsic* difference between the domains; 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 one's interests: Typically, we understand one domain (the "syntactic" one) in terms of an antecedently understood domain (the "semantic" one).[6] 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. (The case of the set-theoretic model implementing a formal theory suggests, in addition, that semantic interpretations can be seen as implementations.)

2. *Implementation in Computer Science*

It is rather surprising how few computer-science texts even try to define 'implementation'. 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*."[7] 'Realization' is left undefined. Taken literally, it means "making real," where 'real' is opposed to 'imaginary' or perhaps 'abstract'; to "realize" X is to establish a real-world, physical correlate of X. This suggests that the physical medium is important.

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."[8] 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. . . . ."[9] Note how *psychological* or *intentional* these realizable things are.

Another computer-science text says a bit more about implementation:[10]

> 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."

A *physical* realization is 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, in medium *M*, of Abstraction *A* (where *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, then implementing the machine-language program in a real computer. As we progress along this "correspondence continuum,"[11] 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."[12] 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.***

3. 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.

3.1 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. (1) One way is to implement it as a finite array A, with $A[0]$ implementing the top. The nature of the implementing medium (e.g., its finiteness) brings along with it certain "implementation details" (concerning, e.g., stack overflow); the ADT Stack "doesn't care" about them (i.e., doesn't—or doesn't *have to*—specify what to do in these cases). (2) Another way is to do everything as before, but let $A[n]$ implement the top. 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.) (3) Or one could use Pascal's *pointer* data type to implement a stack as a "linked list."

Thus, 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?

3.2 The Formal Notion of ADT Implementation

To see what's going on, we need to look at more formal approaches to the definition and implementation of ADTs.[13]

Guttag *et al.*[14] 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."[15] So, implementation—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.)

A more detailed and philosophically sophisticated approach is to be found in Goguen *et al.*[16] 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": An abstraction is either (1) "a mathematical model or description of something," or (2) "the process (or result) of generalizing" or (3) "a concept" considered "independent[ly] of its representation." Examples of (1) are "'abstract machines' as opposed to real hardware" and "abstract

implementation," as "when one uses sets, sequences, or other mathematical entities to model some computational process or structure."[17] 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.[18]

Goguen *et al.* take the third sense of 'abstraction' to be the relevant one for ADTs.[19] The "representation" that such an abstraction is independent of has to do with notation, or the manner in which it is expressed:

> . . . 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 is provided with certain operations, and he only needs to know what they are supposed to do, not how they do it.[20]

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."[21] 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").[22] They define an ADT as "the isomorphism class of an initial algebra in a category" of many-sorted algebras.[23] And they note that "An implementation is necessarily made within a specific framework, such as a particular programming language or machine";[24] i.e., an implementation requires a "framework," i.e., an implementing *medium*.

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."[25] 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 $T_{\Sigma, E}$ be] the specification algebra. The question now is, what relationship between $T_{\Sigma, E}$ and B constitutes an *implementation*?[26]

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) $T_{\Sigma, E}$ to B, and a "congruence" (a family of equivalence relations on (roughly) T's image in B).[27] 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.

4. 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."[28] 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:[29]

(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," in some "medium" *M*, of an "abstraction" *A*. 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. Let us leave the notion unrefined for now, except as that which can be implemented in some medium.

The medium 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 "pseudo-code" and, by "stepwise refinement," fill in the details (e.g., pseudo-code 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 implemen-

tation 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—machine language)—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*: 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."[30] Let's explore these issues.

4.1 Implementation as Individuation

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 *infima 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 *infima 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.[31] 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 *infima 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 individuals are implementations, but not conversely.

4.2 Implementation as Instantiation

Anthony[32] 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 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.[34]

So, for a connectionist architecture to instantiate a classical framework would be for it to have classical properties.

In contrast,

> [w]here *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.[35]

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, a machine-language program might have FA1's properties, but the machine language *itself* might not. As a trivial example, a machine-language program can have records, while the machine language 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."[36] 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 if they are among its primitive data types; otherwise they are a defined ("accidental") feature:

> . . . 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).[37]

Here, 'characteristic' and 'necessary' can be taken to mean "essential." But doesn't a machine-language implementation of Lisp *have* the Lisp function `car`?[38] Since `car` was originally a machine-language instruction ("contents of the address register"), perhaps a better example would be the Common Lisp function `first`.[39] A machine-language implementation of Lisp need not have the Lisp function `first`. It can "simulate" `first`—or implement it?—but it doesn't *have* it; it can do what `first` does without *having* `first`. 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 machine language would be such that the machine language actually *had* identifiable data structures and procedures corresponding to lists, `first`, etc. A weak im-

plementation of Lisp in machine language 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 `first` (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."[40] 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).[41] So, instances and implementations (as Anthony defines them) "are mutually independent."[42]

In any case, 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.

4.3 Implementation as Reduction

Consider a set of axioms for the (non-negative) rationals; i.e., consider the ADT (NonNegative) Rationals. What are some of its implementations? Well, there are the fractions, i.e., the symbol types 0/4, 1/2, 2/3, 1/7, etc. There are the repeating decimals, i.e., the symbol types $0.\bar{0}$, $0.5\bar{0}$, $0.\bar{6}$, $0.\overline{142857}$, etc. There are also certain *constructions* from the integers, e.g., certain ordered pairs of integers: ⟨0, 4⟩, ⟨1, 2⟩, ⟨2, 3⟩, ⟨1, 7⟩, 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, 2/3 and 4/6 are the "same" rational number, as are the repeating decimals $0.1\bar{9}$ and $0.2\bar{0}$ and the ordered pairs ⟨2, 3⟩ and ⟨4, 6⟩. 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. "*Reducibility* . . . is a relation between *theories*; one theory [T] is reducible to another [T_R] if, very roughly, . . . [T's] predicates and claims can be translated into those of" T_R,[43] and T_R is "simpler" or better epistemically grounded than T. 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 1/2 would be to prove a theorem about the following arcane set of sets whose members are integers and sets of integers: $\{\{a, \{a, b\}\} \mid a, b \in \mathbf{Z}^+ \ \& \ 2a = b\}$.[44] 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 1/2 could be implemented as the monstrosity $\{\{a, \{a, b\}\} \mid a, b \in \mathbf{Z}^+ \ \& \ \{\{\{\}, \{\{\}\}\}, \{\{\{\}, \{\{\}\}\}, a\}\} = b\}$. The mind boggles. Is this supposed to be *easier* to understand than '1/2' or '$0.5\bar{0}$'?

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 subset of the rationals is an implementation of the Integer ADT), e.g., the sequence 0/1, 1/1, 2/1, 3/1, ... As a matter of fact, there are several implementations of the integers to be found among

the rationals. 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.

4.4 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).[45]

One major correlation that is a plausible candidate for interpreting implementation is supervenience. "[T]he 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";[46] neither does implementation (§4.3). 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?

4.4.1 Supervenience: An Introduction

Kim[47] 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,"[48] even though "there is no relationship of definability or entailment between the two families" of properties.[49] 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."[50] 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 *:[51] 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 $M^{\#}$).[52] 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 $M^{\#})$.[53] 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* $D =_{df} \Box((\forall P_N \in N^{\#})[P_N(d) \leftrightarrow P_N(d')] \rightarrow (\forall P_M \in M^{\#})[P_M(d) \leftrightarrow P_M(d')])$.

Kim presents an argument that reducibility and definability entail supervenience.[54] 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 bi-

conditionals? 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 law-like.[55] This is surely true for implementation in the N-to-M direction.[56] Is it true in the M-to-N direction?[57] 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;[58] (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.[59]

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 (intensionally) 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. Then it is the implementing medium that has both *A*-properties and *I*-properties.[60] 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 sees supervenience as a very general version of the family of concepts that includes reducibility, etc.[61] 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."[62] 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*.[63]

and he points out that "Both relations are transitive, reflexive, but neither symmetric nor asymmetric."[64] 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.[65] So it looks as if supervenience is *not* implementation.

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 examples of implementation, but not *vice versa*. The single best "interpretation" of implementation seems to be that of semantic interpretation: *I* is an implementation, in medium *M*, of Abstraction *A* 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.[66]

William J. Rapaport

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

NOTES

1. Cf. Soare 1996; Rapaport, 1998.
2. See Rapaport 1988, 1995; Smith 1997.

3. Cf. Rapaport 1995 for a more elaborate survey.
4. Cf. Bunn, forthcoming.
5. Cf. Rapaport 1988, 1995.
6. Cf. Rapaport 1995.
7. Marcotty & Ledgard 1986, p. 8; my boldface.
8. Simpson & Weiner 1989, Vol. 13, p. 272.
9. *Op. cit.*, p. 277.
10. Hayes 1988, p. 47; my boldface.
11. Smith 1987; cf. Rapaport 1995.
12. Hayes 1988, p. 209, my italics.
13. A good survey is Morgado 1986.
14. Guttag *et al.*, 1978, p. 61.
15. *Ibid.*
16. Goguen *et al.* 1978.
17. All quotations are from Goguen *et al.* 1978, pp. 82–83.
18. Cf. the "muddle of the model in the middle": Wartofsky 1979: xiii–xxvi, Rapaport 1995.
19. Cf. Goguen *et al.* 1978, p. 81.
20. *Op. cit.*, p. 83; cf. Parnas 1972.
21. Goguen *et al.* 1978, p. 83.
22. *Ibid.*
23. *Op. cit.*, pp. 88, 90.
24. *Op. cit.*, p. 135.
25. *Ibid.*
26. *Op. cit.*, p. 136.
27. *Op. cit.*, p. 138.
28. Simpson & Weiner 1989, vol. 7, p. 721.
29. *Op. cit.*, p. 722.
30. Guttag *et al.* 1978, p. 62.
31. Cf. Castañeda 1975.
32. Anthony 1991.
33. *Op. cit.*, p. 325; my italics.
34. *Ibid.*
35. *Ibid.*
36. *Ibid.*
37. *Op. cit.*, p. 326; my italics.
38. 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.
39. See the previous note.
40. *Op. cit.*, p. 326.
41. Cf. Shapiro 1979; Shapiro & Rapaport 1987, 1992.
42. Anthony 1991, p. 326.
43. Smith 1991, p. 280, n. 39.
44. 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" $\{\langle a, b\rangle \mid a, b \in \mathbf{Z}^+ \ \& \ 2a = b\}$. So, the rational 1/2 "is" $\{\{a, \{a, b\}\} \mid a, b \in \mathbf{Z}^+ \ \& \ 2a = b\}$.
45. Cf. Rapaport 1986, 1988, 1995.

46. Smith 1991, p. 280, n. 39.
47. Kim 1978.
48. *Op. cit.*, p. 149; my italics.
49. *Op. cit.*, pp. 149–50.
50. *Op. cit.*, p. 150. But cf. the quotation from pp. 153–54, below.
51. *Op. cit.*, p. 152, col. 1.
52. *Ibid.*
53. That, at least, is what Kim *says*; but doesn't he mean M^* and N^*? Perhaps not. Cf. Kim 1978, p. 153, col. 1.
54. *Op. cit.*, p. 152, col. 1.
55. *Op. cit.*, p. 152, col. 2.
56. *Op. cit.*, p. 152, col. 1.
57. *Op. cit.*, p. 152, col. 2.
58. But cf. the quotation from Kim 1978, p. 150, above.
59. *Op. cit.*, pp. 153–54.
60. If a stack is implemented as a Lisp list, then a stack property (an *A*-property) might be the relation `top(push(e,s)) = e`, while the corresponding Lisp-list property (an *I*-property) would be `(first (setf s (cons e s))) = e`.
61. Kim 1979, pp. 43–44.
62. *Op. cit.*, p. 40.
63. *Op. cit.*, p. 49.
64. *Ibid.*
65. Cf. Rapaport 1995.
66. 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

Anthony, Michael V. (1991), "Fodor and Pylyshyn on Connectionism," *Minds and Machines* 1: 321–41.

Bunn, James H. (forthcoming), "Universal Grammar or Common Syntax?: A Critical Study of Jackendoff's *Patterns in the Mind*," *Minds and Machines*.

Castañeda, Hector-Neri (1975), "Individuals and Non-Identity: A New Look," *American Philosophical Quarterly* 12: 131–40.

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.

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.

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

Kim, Jaegwon (1978), "Supervenience and Nomological Incommensurables," *American Philosophical Quarterly* 15: 149–56.

_____ (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, MN: University of Minnesota Press): 31–49.

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

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

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

Parnas, David (1972), "A Technique for Software Module Specification with Examples," *Communications of the Association for Computing Machinery* 15: 330–36.

Rapaport, William J. (1986), "Searle's Experiments with Thought," *Philosophy of Science* 53: 271–79.

_____ (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, CA: Academic Press, 1994): 225–73.

_____ (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).

_____ 1998, "How Minds Can Be Computational Systems," *Journal of Experimental and Theoretical Artificial Intelligence* 10: 403–13.

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

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

Sellars, Wilfrid (1955), "Some Reflections on Language Games," in *Science, Perception and Reality* (London: Routledge & Kegan Paul, 1963): 321–58.

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.

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.

Shapiro, Stuart C., & Rapaport, William J. (1992), "The SNePS Family," *Computers and Mathematics with Applications* 23: 243–75; reprinted in Fritz Lehmann (ed.), *Semantic Networks in Artificial Intelligence* (Oxford: Pergamon Press, 1992): 243–75.

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

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

_____ (1991), "The Owl and the Electric Encyclopedia," *Artificial Intelligence* 47: 251–88.

_____ (1997), "One Hundred Billion Lines of C++," *CogSci News* (Bethlehem, PA: Lehigh University Cognitive Science Program), vol. 10, no. 1 (Spring): 2–6

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

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