One especially nice feature of this approach is that it does not build in any notion of causality, and therefore can capture the sort of mutual interdependence that occurs, for example, in electrical circuits. This contrasts with models like automata in which a causal dependence of the next state on the current state and current input is built into the model. It is also consistent with philosophical ideas like mutual causation and interdependent origination[4], which go beyond naive reductionist causality. But it was (and still is) disappointing to me that so few people felt any need for concepts and theories of such generality; they seem happy to have (more or less) precise ideas about specific systems or small classes of systems, with little concern for what concepts like system, behavior and interconnection might actually mean. Still, this categorical GST has had a significant indirect impact on computer science: its application to the Clear and OBJ module systems influenced some important programming languages, including Ada, ML and C++ (see Section 2.6).

A general theory of objects based on sheaf theory [40] arose from this work, and has been applied (for example) to the semantics of concurrent systems, including concurrent object based languages [62], hardware description languages [166], and semantics for object based concurrent information systems [23]. Sheaves can express the kind of local causality combined with global non-determinism that characterizes many different kinds of model, from partial differential equations to automata. They can capture not only variation over time, but also over space and over space-time [62]; and they can embrace the incompleteness of observation that is characteristic of all real empirical work. This can be helpful at the philosophical level in dispelling the illusion that models fully capture reality (see the discussion in Section 4.1). It can also be useful technically, for example in capturing the way that the behavior of distributed concurrent systems depends only on local interactions. There is now a slow but steady stream of research on sheaf theory in computer science, though there is not as yet a coherent community.

In the early 1970s, I formulated the minimal realization of automata as an adjoint functor [38]; this soon evolved into much more general results about the minimal realization of machines in categories, which gave a neat unification of system theory (in the sense of electrical engineering) with automaton theory [36]. I consider this a major vindication of the categorical approach to systems.

2.3 Abstract Data Types and Algebraic Semantics

The history of programming languages, and to a large extent of software engineering as a whole, can be seen as a succession of ever more powerful abstraction mechanisms. The first stored program computers were programmed in binary, which soon gave way to assembly languages that allowed symbolic codes for operations and addresses. FORTRAN began the spread of "high level" programming languages, though at the time it was strongly opposed by many assembly programmers; important features that developed later include blocks, recursive procedures, flexible types, classes, inheritance, modules, and genericity. Without going into the philosophical problems raised by abstraction (which in view of the discussion of realism in Section 4 may be considerable), it seems clear that the mathematics used to describe programming concepts should in general get more abstract as the programming concepts get more abstract. Nevertheless, there has been great resistance to using even abstract algebra, let alone category theory.

One of the most important features of modern programming is abstract data types (hereafter, **ADT**s), which encapsulate some data within a module, providing access to it only through operations that are associated with the module. This idea seems to have been first suggested by David Parnas [155, 156] as a way to make large programs more manageable, because changes will be confined to the inside of the module, instead of being scattered throughout the code. For example, if dates had been encapsulated as an ADT, the so-called "year two thousand problem" would not exist. Not all increases in abstraction make programming easier; an abstraction must match the way programmers think, or it won't help. This may explain why ADTs have been more successful than higher order functions.

In the early 1970s, there was no precise semantics for ADTs, so it was impossible to verify the correctness

[4]The concept of interdependent origination goes back over 2,500 years to the Buddha; in the Pali language, it is called *paṭicca-samuppāda* [12]. This kind of thinking can also be found in much contemporary AI, e.g., the robotics of Rodney Brooks, which is intended to be fast and cheap, because it doesn't require any central control (cf. the wonderful documentary movie *Fast, Cheap and out of Control*, directed by Errol Morris).

of an implementation for a module, or even to formulate what correctness means. Initial algebra semantics provided the first rigorous formulation of these problems, with solutions that were useful, although they have been improved (see Section 2.10). Initial algebra semantics was born in [41], which (among other things) formulated (Knuthian) attribute semantics as a homomorphism from an initial many sorted syntactic algebra generated by a context free grammar, to a semantic algebra[5]. The step to ADTs was facilitated by my realization that Lawvere's characterization of the natural numbers as an initial algebra [132] could be extended to other data structures [105, 106]. As a young researcher at this time, I was really shocked by the attempts of certain senior colleagues to reconfigure the history of this period to their own advantage; this is why I wrote the paper [51].

What really pleased me was the neat parallel established between Emmy Noether's insight that algebra is the study of sets with structure given by operations, and David Parnas's insight that modules should encapsulate data with operations; more than that, the algebraic approach established an equivalence between abstractness in ADTs and abstractness in algebra[6]; furthermore, equations among operations came into specifications of ADTs the same way as in abstract algebra.

It also seemed splendid that computability properties worked out so well: an algebraic version of the Turing-Church thesis says that an algebra is computable iff it is a reduct of a finitely presented initial algebra with an equationally definable equality; these are also the algebras for which so-called "inductionless induction" proofs are valid [47]. Moreover, an algebra is: semicomputable iff it is a reduct of a finitely presented initial algebra; cosemicomputable iff it is a reduct of a finitely presented final algebra; and computable iff a reduct of a finitely presented algebra that is both initial and final. The reason that the computability notion associated with Scott-style denotational semantics doesn't work for algebras is explained in [46]. This field was pioneered in a series of papers by Jan Bergstra and John Tucker, surveyed in [150], which also explains basic many sorted algebra and abstract machines. Some conjectures from [150] were solved with Meseguer and Moss in [153]. The computational side of ADTs includes term rewriting, which featured in early drafts of [106]. Initial algebra semantics has also been used in linguistics, to explicate to the notion of compositionality [122].

Meseguer and I studied the rules of deduction for many sorted algebra in [94]. This paper surprised the community by showing that the naive generalization of the usual unsorted rules (as previously used in the ADT literature) is unsound. We gave a sound and complete set of rules, and showed that the unsorted rules did work for certain signatures; the difficulties involve implicit universal quantification over empty sorts. Complete rules of deduction for many sorted conditional equational logic are given in [92]. The first rigorous proof of correctness for the inductionless induction proof technique that was originally suggested by David Musser, is given in [47]. The formulation of inductionless induction in [47] is more general than in some later work, which was restricted to just constructors; [47] also pointed out that the essence of inductionless induction is "proof by consistency," and gave the simple but fundamental result that for an equational specification that is canonical (i.e., terminating and Church-Rosser) as a set of rewrite rules, the normal forms of ground terms form an initial algebra. This result justifies term rewriting as an operational semantics for initial algebra semantics.

ADJ later extended initial algebra semantics to continuous algebras (at about the same time as Maurice Nivat) and continuous algebraic theories [110], and then to rational algebraic theories [173]. This inspired Meseguer and me to generalize to an arbitrary category, getting *initial model semantics* (see Section 2.9).

2.4 Specification

In the early seventies, most theoretical research concerned the semantics of programs and programming languages, and the verification of programs. There was little or no work on specification, modules, or

[5]This built on an approach to many sorted algebra developed for my course Information Science 329, Algebraic Foundations of Computer Science, first taught in 1969 at the University of Chicago, including the now familiar use of indexed sets, the word 'signature' with symbol Σ, and its formal definition.

[6]This is because any two initial objects in a category are isomorphic; hence we speak of "the" abstract data type of a specification in exactly the same way that we speak of "the" initial algebra of a variety. Each is determined up to isomorphism, and the fact that each an "abstract algebra" and an "abstract data type" are isomorphism classes of algebras expresses their independence of representation.