MODEL THEORETIC ISSUES IN THEORETICAL COMPUTER SCIENCE, PART I: RELATIONAL DATA BASES AND ABSTRACT DATA TYPES

by

J.A. Makowsky

Technical Report No. 270

April 1983
MODEL THEORETIC ISSUES IN
THEORETICAL COMPUTER SCIENCE, PART I:
RELATIONAL DATA BASES AND ABSTRACT DATA TYPES.

J.A.Makowsky

Department of Computer Science,
Technion - Israel Institute of Technology,
Haifa, Israel


Table of Contents:
Introduction.
1. Abstract model theory and computer science.
   1.1. From syntax to semantics and back.
   1.2. Finding axioms.
   1.3. Comparing logics.

2. Data base theory.
   2.1. What it is all about.
   2.2. Safety (definiteness, domain independence).
   2.3. Typed dependencies.
   2.4. Implicational dependencies.
   2.5. Decision problems.
   2.6. Query languages.
   2.7. Conclusions and some open problems.

   3.1. Introduction.
   3.2. The axiomatic framework.
   3.3. A complete specification language for rich semantical systems.
   3.4. Typical models and initial algebras.
   3.5. A complete language for semantic systems which admit initial semantics.
   3.6. Relevance for specification of abstract data types.
   3.7. A word on other applications.

References

Supported by the Swiss National Science Foundation Grant No. 82.820.0.80
Introduction

The following paper is an account of experiences I had in several attempts to capture problems posed in computer science, or more precisely by computer scientists. In most of the cases the computer scientist already thought that logic might help in stating problems more precisely, and ultimately, also in solving them, though they were usually suspicious about the impact such solutions would have on their direct practical involvement with programming, program analysis, program design or program verification.

Maybe a word on impact of foundational studies on applied science and engineering is needed here: Most electricians are not aware how much 19th century physics has contributed in making the portability of electrical appliances possible. The fact that two or three numeric parameters (voltage, power and the number of cycles in alternating current) contain all the information needed to decide, whether a given appliance can be used by plugging it into a given outlet, has become too common in every day life to be reflected upon. To remember that many years of research were needed to clarify this situation, is by now safely forgotten.

Ultimately the problems posed by computer scientist to logicians, or for that matter to any one willing to spend time on foundational questions, is similar. What are the parameters needed to ensure portability and reliability of software?

Needless to say, we are still very far from satisfactory answers. The current progress in technology even prevents computer manufacturers from reaching agreements, as they were reached rather quickly, say, by manufacturers of phonographs and, to some extent, videotape systems, on speed and size of the records (tapes) to be produced. But a deeper reason behind the problem consists in the absence of a definite model of the real world, here the programming environment. Though models of “computability” have been sufficiently clarified for deterministic sequential algorithms, provided their task can be unambiguously specified in some form of “natural scientific language”, it is much less clear what “specification”, “implementation” and “correctness” should mean. The problems involved are not exclusively problems of computer science. Any large scale design and implementation of a big organisational complex, from industrial to social engineering, touches upon the same fundamental questions. The only difference with computers stems from the fact, that they execute programs very quickly, and programs, which are used only once or a few times, become very soon obsolete. But we are generally inclined to expect that the time needed to develop a program stays within proportions to the time it runs and remains useful. This leads some to think, that also the foundational questions can be solved quickly. However a short glance at the history of mathematics shows us, that something like fifty years were spent till the basic notions of, say, point set topology were safely established and had their impact not only on mathematicians but also physicists and practitioners of
In the following paper I will try to explain how I have learned to view certain problems in the foundations of computer science and I will do this with three apparently different aspects of current computer science research: Data base theory, algebraic specification of abstract data types and algorithmic logic. As I will try to show, they have much more in common than widely believed, at least when looked at from the point of view of "abstract model theory". But I am fully aware that the challenge of applied science is not met by declaring that some ready made theory captures all its problems. This is never so. The difference between pure and applied, say, in differential equations, consists rather in the motivation of the results than in the results itself. The pure mathematician is content with knowledge which contributes to his understanding of the internal problems of differential equations as such, and solving a particular one is seen by him as a challenge of his general understanding. To the physicist, most of the work actually consists in justifying his particular differential equation, its parameters and its solutions in terms of his physical problem, and a large part of the work of good applied mathematics consists in reaching an understanding between the two perspectives. In computer science, we meet the same situation and both sides are often tempted to underestimate the work involved in listening to each other. This paper is also an attempt to illustrate this work.

For reasons of space (in this proceedings) and time (the Damocles sword of the deadlines), the paper had to be cut into two parts. The first deals with data base theory and specification of abstract data types, and the second one with various approaches to semantics of programming languages. The first part also includes a general introduction on abstract model theory and its potential use in theoretical computer science. The second part [Makowsky 1983] will also include an expository chapter of some more technical parts of abstract model theory.

Here is an outline of both parts: In chapter 1 we try to give a description of what abstract model theory is all about, and how it is connected to the fundamental questions of computer science cited above.

In chapter 2 we try to exemplify this in the case of data base theory. As it turns out there are various intimate connections between finite model theory and data base theory, which have led people to think that either data base theory is just undergraduate logic or that the logicians try to sell it as such. But the real problems in any applied science are neither defined by their mathematical difficulty nor by the methodologies used to solve them, but rather by the questions they try to answer. Data base theory tries to answer the questions about design, design criteria, optimization and specification of data bases and their queries. This chapter could not have been written without the patience of C.Beeri, M.Vardi and A.Zvieli.
In chapter 3 we turn our attention to the problem of specifying abstract data types. There are various approaches to this problem, but the most successful, at least in terms of fashion, is the one called algebraic. Here we find an interesting interplay between category theory and universal algebra, which has led people to think that this is just disguised "abstract nonsense", but again, as above, the problem we address here is the clarification of concepts such as data type, implementation, specification, modular programming and their ramifications and only a close, sympathetic analysis of these problems can lead to satisfactory answers. The results reported here are joint work with B.Mahr.

As it turns out, there is a common theme in these last two chapters: In both universal Horn formulas play an eminent role. In the last section of chapter 3 we try to give some explanation of this phenomenon. It seems to support some of the arguments put forward by the proponents of logic programming as the programming style appropriate for the fifth generation computers. However, some recent complexity results, such as [Itai-Makowsky 1982] still nourish some scepticism with respect to the unrestricted use programming languages like PROLOG.

The remaining two chapters form part two ([Makowsky 1983]): In chapter 4 we turn to the technical parts of abstract model theory as we see them fit the needs of various branches of program semantics and program verification. We attempt to give a general definition of predicate transformers, as they appear in the context of program correctness. The definition is parallel to the definition of generalized quantifiers, which will turn out to be a special case. On the basis of a set of predicate transformers one can build various algorithmic logics, of which again the classical examples of dynamic logic, process logic and others are special cases.

In chapter 5 we turn our attention to program correctness and programming logics. We use the various logics from the previous chapter to introduce a new type of semantics, which in contrast to operational or denotational semantics, maps programming languages into subsets of logics. Here the meaning of a program is the set of all statements in a predicate transformer logic which are true about it. This is clearly not new as such, but has never been defined in a general context. One of the advantages of such a general approach is, that this allows us also to compare various approaches to program semantics which hitherto were considered incomparable. This last chapter is to report about work which is still in progress, mainly in collaboration with N.Francez and S.Katz.

I would like to thank the Swiss National Science Foundation, who supported me generously during the two years in which the material presented came into being. I would like to thank also E.Engeler and E.Shamir who encouraged me to look into foundational problems in Computer Science and to C.Beeri, A.Meyer and V.Pratt, whose interest and criticism in early stages of the work was extremely stimulating. I would
also like to thank R.Fagin and J.Thatcher, who read and commented the almost final version.

A last note on the references: There are over one hundred titles listed as I saw them fit my presentation. I did not try to give historic remarks, nor did I attempt completeness. I tried to give the reader pointers to the vast literature, as I came across it during my random walk in the world of theoretical computer science. It serves as a basis for further backtracking: The transitive closure of this reference list surely covers much.
1. Abstract model theory and computer science.

1.1 From syntax to semantics and back.

In the early days of modern logic, logic was perceived mainly syntactically. Propositional logic, first order logic and second order logic were given as formal languages and a main topic of research was the study of deduction rules and proof systems. There are various philosophical, sociological and even political reasons for this, usually subsumed under the name "Hilbert's program". G. Kreisel has written extensively about Hilbert's program and the way it failed. From his analysis in [Kreisel 1968, 1970] he drew several conclusions relevant for computer science which inspired the theses of R. Statman [Statman 1974] and C. Goad [Goad 1980]. The former added a new dimensions to our understanding of the complexity of proofs and the latter used his experience gained in proof theory to speed up the synthesis of special purpose programs for hidden surface elimination, [Goad 1982].

Hilbert's program wanted to reduce mathematics, and therefore all exact sciences, to the formal (or, as we would say today: algorithmic) manipulations of symbols. The ultimate hope behind this was, to find general purpose algorithms, which would solve all formally stated problems. As we know today, Godel showed that this is impossible. But at the same time modern semantics was born. The fashion had changed, and instead of the "God given" Natural Numbers, Tarski and his contemporaries moved to accept naive set theory as the basis of mathematics and proposed to explain logic in terms of set theory. The meaning of logical formulas was explained in terms of structures, relations, functions and in the case of first order logic this was justified by the celebrated completeness theorem.

One of the corollaries of the completeness theorem is the compactness theorem, which was extended to uncountable sets by Mal'cev in 1936 and independently by Henkin in 1949. Among the many consequences of the compactness theorem is the existence of various "non-standard" models of arithmetic and analysis, which later led to a very fruitful branch of logic called non-standard analysis, which was first pursued by A. Robinson and led to various impressive results in analysis, Banach space theory, the theory of Brownian motion and even mathematical economics. But one of the first non-trivial applications of the compactness theorem was the characterization of the universal first order formulas in terms of an algebraic preservation property:

A first order formula $\varphi$ is logically equivalent to a universal formula if and only if $\varphi$ is preserved under substructures.

This result was followed by an intensive program exploring the relationship between semantic properties of formulas and syntactic characterizations of such formulas. Motivated by algebraic practice the notion of substructure was replaced successively by unions of chains, products, reduced products, factors and many others and sufficient experience was gained to delegate this direction of research to the level of master theses and difficult exercises. The mathematical tools used to solve such
problem, if there is a clean solution, usually are interpolation theorems, ultraproduc
ts and Back and Forth arguments. In [Chang-Keisler 1973] the reader may find what
ever is known in this direction. But there is another way in looking at this program: We
could reverse the problem and start with any syntactically defined class of formulas
together with their meaning functions and ask for a characterization of this class in
terms of the preservation properties it has. Looked at it this way, what we really are
asking for is giving meaning to syntactic categories. It is this aspect of preservation
theorems which I think is relevant to foundational questions in computer science.
Very often the computer scientists start with syntactic restrictions and later try very
hard to remove them, without understanding their significance. But, as will be shown
in chapters 2 and 3, those restrictions, originally imposed for technical reasons, can
be characterized by preservation properties, which show that they are intimately con-
nected with the implicit assumptions the computer scientists have made.

The use of powerful set theoretic methods led also to another development. Already in
the fifties Mostowski in Warsaw, Engeler in Zurich and Tarski and Henkin in Berkeley
started to look at various generalizations of first order logic involving infinitary con-
structs and generalized quantifiers and an abundance of logics appeared. It was
Engeler, however, who first noticed the possible relevance of infinitary logics to com-
puter science ([Engeler 1967,1970]). This has since led to the development of dynamic
logic, and we shall return to this topic in chapter 4 and 5 of this paper. In the sixties
much of model theory was generalized to those newly discovered logics, and, based on
earlier work by Mostowski, Lindstrom defined an axiomatic framework, sometimes
called abstract model theory or higher model theory, in which we can study logics in
general, compare their expressive power and prove characterization theorems for log-
ics in terms of their model theoretic properties. The latter has very striking parallels
with the above mentioned preservation theorems both in content as well as in metho-
dology. In the rest of this chapter we shall briefly describe this framework and give
some key results, which we will apply in chapter 4 and 5 to the study of various
dynamic logics. An account of the state of art in abstract model theory can be found
in the forthcoming book [Barwise-Feferman 1983], and an introductory survey in
[Flum 1975].

The purpose of abstract model theory can be summarized as follows:
We want to be able to quantify over all possible logics satisfying certain properties and
prove theorems about them. The theorems we want to prove can be
(A) characterization theorems.
(B) presentation theorems.
(C) theorems relating various properties of logics.
Examples for (A) are the Lindstrom theorems, ([Lindstrom 1969]), characterizing first
order logic in terms of the Lowenheim-Skolem theorem together with various proper-
ties such as compactness or axiomatizability. The latter has been closely analyzed for
its usefulness in computer science in [Manders-Daley 1983] and in [Makowsky 1980].
Examples for (B) are Birkhoff's theorem characterizing the varieties as the
equationally definable classes of algebras (cf. [Graetzer 1979]), Mal'cev's characterization of the quasi-varieties (cf. [Mal'cev 1971]) and Cudnovskii's theorem that every class of structures closed under substructures can be axiomatized by a class of infinitary clauses (cf. theorem 1 in chapter 3 and [Cudnovskii 1968]). In some sense the result in [Meyer-Parikh 1981], showing that most dynamic logics for finitely branching programs can be embedded in the recursive part of countably infinite logic, also fits this category. An example for (C), finally, is that axiomatizability implies recursive compactness or that for countable logics the amalgamation property is equivalent to compactness. The former is a corollary of the Lindstrom theorems and the latter is in [Makowsky-Shelah 1983].

The use of abstract model theory lies in its limitative character. It tells us that certain requirements are incompatible or entail other limitations. It can tell us to what extent seemingly different approaches are nevertheless the same. Or it can give us a framework in which we can precisely compare concepts which hitherto appeared incomparable.

1.2 Finding axioms.

Our first problem is to find axioms for logics. Logics will consist of quadruples \( L = (T, \text{Str}, \text{Sim}, \models) \), where \( T \) is a class of vocabularies or signatures and \( \text{Str} \) is a function mapping every \( \tau \in T \) into a subclass \( \text{Str}(\tau) \) of all structures of vocabulary \( \tau \) such that if \( \tau, \tau' \in T, \tau \subseteq \tau' \) then \( \text{Str}(\tau) \subseteq \text{Str}(\tau') \). Here we assume that in \( T \) we have a partial order denoted by \( \subseteq \). In all the cases we consider the elements of \( T \) are sets of symbols and \( \subseteq \) just is the subset relation.

The usage of the term vocabulary for what is called similarity type (or signature or even language) seems to capture what we really have in mind. The vocabulary is the most elementary part of logic, and it determines about what we will talk. In the case of first order logic it consists just of sets of relations symbols, function symbols or constant symbols, together with their arities, and in the case of many sorted logic, with their sort specifications. In other cases it may also specify variables to be second order, or make certain distinctions between logic with or without equality, or other special symbols. Sometimes it is convenient to think of \( T \) as a category of vocabularies rather than just a set or a class. This is especially the case when want to consider vocabularies which are more complicated than usually and we have to define the partial order on \( T \) in a more complex way. But these cases are still not very well developed.

Now given \( T \), the class \( \text{Str}(\tau) \) tells us which structures of vocabulary \( \tau \) we are interested in. This may, in logic, often comprise all the \( \tau \)-structures, but in applications we very often impose restrictions. In chapter 2 we shall see that for data base theory we only consider finite structures, or finite reducts of expansions of the standard model of arithmetic, and in chapter three only certain countable structures, the reachable structures are of interest. Finally in chapters 4 and 5 various more complicated structures will enter the picture, including certain models of tense logic, probability logic etc. Again it may be convenient to think of \( \text{Str}(\tau) \) as subcategories of a big
category $\text{Struct} = \bigcup_{\tau \in T} \text{Str}(\tau)$ and of a functor mapping $T$ into $\text{Struct}$. We refer the reader to [Barwise 1974] for a detailed presentation. Also $\text{Fml}$ is a function which maps every $\tau \in T$ into a set of objects called formulas. Again we require that for $\tau, \tau' \in T, \tau \subset \tau'$ we have that $\text{Fml}(\tau) \subseteq \text{Fml}(\tau')$. When choosing the set of formulas, we have to bear in mind to contradicting aspects: We want to say much about our structures, certainly as much as we need in our particular context. But we do not want to say too much, because we want to keep our model theory out of the difficulties of full second order logic.

Finally $|= $ is a relation on $\text{Str}(\tau) \times \text{Fml}(\tau)$, which satisfies certain axioms:

**Isomorphism Axiom:** If $A, B \in \text{Str}(\tau), \varphi \in \text{Fml}(\tau)$ and $A \models B$ then $A |= \varphi$ iff $B |= \varphi$.

**Reduct Axiom:** If $\varphi \in \text{Fml}(\tau), \tau \subset \tau'$ and $A \in \text{Str}(\tau')$ then $A |= \varphi$ iff $A \models \tau |= \varphi$.

**Renaming Axiom:** Let $\sigma, \tau \in T$ and $\rho: \tau \to \sigma$ be a renaming, i.e. an isomorphism in the category of vocabularies. Then for each $\varphi \in \text{Fml}(\tau)$ there is $\varphi^\sigma \in \text{Fml}(\sigma)$ such that for all $A \in \text{Str}(\tau)$ we have that $A |= \varphi$ iff $A^\sigma |= \varphi^\sigma$.

These axioms do not require too much. All examples which we shall encounter in this paper satisfy them. From the theorems in abstract model theory cited in the previous section, however, only example (B) can be proved with these axioms alone. In chapter 3 we shall use similar axioms to axiomatize the behaviour of sets of formulas rather than formulas.

What makes abstract model theory into a theory are various additional closure properties, which we impose on the formulas, or rather on their models $\text{Mod}_*(\varphi) = \{ A \in \text{Str}(\tau) : A |= \varphi \}$. Many model theoretic properties of various logics can be stated by only referring to the model classes $\text{Mod}_*(\varphi)$ definable by their formulas. The compactness theorem and the Lowenheim-Skolem theorem are among them, and also various definability theorems. But here are some of the closure axioms:

**Atomic Axiom:** For every $\tau \in T$ the usual $\tau$-atomic formulas are contained in $\text{Fml}(\tau)$.

**Basic Axiom:** For every $\tau \in T$ the usual $\tau$-basic formulas (i.e. atomic and negated atomic formulas) are contained in $\text{Fml}(\tau)$.

**Boolean Axiom:** $\text{Fml}(\tau)$ is closed under the boolean operations $\land, \lor, \neg$ with their usual meaning, i.e. their model classes are defined by intersection, union and complement respectively.

**Quantification Axiom:** $\text{Fml}(\tau)$ is closed under existential quantification $\exists x$ with its usual meaning.
The next two axioms assume some knowledge of the structure of the formulas if we want to state them naturally. They are the relativization axiom and the substitution axiom. We will discuss them in more detail in chapter 4. For the first part of the paper their exact definition is irrelevant. Examples of logics are first order logic, infinitary logics, logics with generalized quantifiers etc. All the classes of dependencies in chapter 2 can be viewed as logics (though without all the closure properties) and in some sense also the semantical systems of chapter 3.

Behind the choice of closure properties lies the problem of iteration under various formation rules for formulas, in other words the choice of primitives for our logics. Already in the early days of infinitary logics did Kreisel point out in [Kreisel 1968] that we need definability criteria to evaluate such choices, rather than just adding various constructs ad libitum. He advocated a line of research which not only led to unifying results for infinitary logics and generalized recursion theory but also to a deeper insight in general. It led to the very rich theory of admissible sets, as presented in [Barwise 1975].

The corresponding problem we face in computer science has not even been formulated generally. In data base theory only [Chandra 1981] questions the choice of programming primitives and [Chandra-Harel 1980] define general criteria for query languages. We shall study the latter in section 2.7 and show that in this case "Kreisel's program" can be followed to a large extent. For specifications of abstract data types [Burstall-Goguen 1983] and [Mahr-Makowsky 1983] attack this problem. In the second part of the paper we shall outline what can be done for semantics of programming languages, but we are still far from a general understanding.

When we want to apply the framework of abstract model theory to foundational problems in computer science, we observe quickly that what we hope to be logics are usually not closed under all the closure operations we have mentioned above. The striking example here is Hoare logic, which consists of statements about programs of a particular form, the correctness statements, but is not closed under any iteration or boolean operation. This leads to an abundancy of logics which are hard to compare, cf. [Meyer-Tiuryn 1981] and [Meyer 1980]. The reasons for the absence of closure properties are sometimes not clear, but in other cases motivated by practical experience. Logics are usually closed under substitutions of predicate symbols by formulas, but in programming some formulas occur as test in programs, and those should remain simple, and it is clear that we do not allow termination statements of other programs to occur as test. The choice of the correct closure properties for logics affects the applicability of results from abstract model theory very much. Sometimes, however, the absence of closure properties can be compensated by a weakening of the theorems. Often a theorem states that every formula $\varphi$ in a given logic satisfying certain conditions is equivalent to another formula in a different logic. In the absence of the Boolean axiom, this may be conveniently rephrased by stating that $\varphi$ is equivalent to a boolean combination of such formulas. In more complicated cases, however, we have
Definitions:

(i) Let \( \varphi \in \text{Fml}(\tau \cup \{R\}) \) \( R \not\in \tau \) be a formula and \( \sigma \subseteq \tau \). We say that \( \varphi \) \textit{defines} \( R \) \textit{implicitly over} \( \sigma \) if:

Every \( \sigma \)-structure \( A \) can be expanded to a \( \tau \cup \{R\} \)-structure \( A' \) such that \( A' = A \) and given two structures \( A, B \in \text{Str}(\tau \cup \{R\}) \) with \( A = B \) such that \( A \models \varphi, B \models \varphi \) such that \( A \models \sigma = B \models \sigma \), then \( R_A = R_B \), i.e. R is uniquely determined by \( \varphi \) and \( \sigma \).

(ii) Let \( \varphi \in \text{Fml}(\tau \cup \{R\}) \) be a formula which defines \( R \) implicitly over \( \sigma \), \( R \) \( n \)-ary. We say that \( \psi(v_1, v_2, \ldots, v_n) \in \text{Fml}(\sigma) \) with \( n \) free variables \textit{defines} \( R \) \textit{explicitly}, if for every \( A \in \text{Str}(\tau) \) with \( A = A \) we have that \( A \models \forall v_1, v_2, \ldots, v_n (R \leftrightarrow \psi(v_1, v_2, \ldots, v_n)) \).

With these definitions we can state an even stronger closure property:

\textbf{\( \Delta \)-closure Axiom:} Every implicitly defined relation has an explicit definition.

Examples:

(i) (Many-sorted) first order logic satisfies all the closure axioms. The \( \Delta \)-closure is a variant of Beth’s definability theorem first stated in [Feferman 1974].

(ii) First order logic without function symbols and with all structures finite, as we shall use it for database theory in chapter 2, does not satisfy the \( \Delta \)-closure axiom, as pointed out in [Hajek 1976].

(iii) Note that \( \Delta \)-closure is a stronger property than closure under substitution.

(iv) It is this \( \Delta \)-closure property which made various ideas of [Kreisel 1968] more precise. Kreisel’s work led to the definition of \textit{admissible sets}, and H.Friedman showed a deep connection between \( \Delta \)-closed logics and logics built on admissible sets, cf. [Makowsky-Shelah-Stavi 1976].

(v) An interesting application of Beth’s theorem to database decomposition problems may be found in [Vardi 1982].

More examples will be studied in chapter 4.

1.3 Comparing logics.

When we want to compare logics, we want to compare their expressive power, i.e. what subsets or relations of its structures are definable. We say that a logic \( L_1 \) is \textit{reducible} to a logic \( L_2 \) if every formula of \( L_1 \) can be translated into a formula of \( L_2 \). Again our notion of comparability will depend on the various closure properties the logics in question have. For positive results we are usually interested in the highest possible degree of precision on the nature of this translation, whereas for negative results, on the contrary, we prefer ample freedom. Let us propose some definitions:
Definitions: Let $L_i = (T_i, \text{Str}_i, \text{Fml}_i, \models_i)$ be logics for $i = 1, 2$.

(i) $L_1$ is explicitly reducible to $L_2$, if $T_1 \subseteq T_2$, for each $\tau \in T_1$, $\text{Str}_2(\tau) \subseteq \text{Str}_1(\tau)$, and for every $\varphi \in \text{Fml}_1(\tau)$ there is $\psi \in \text{Fml}_2(\tau)$ such that

$$\text{Mod}_\tau(\varphi) \cap \text{Str}_2(\tau) = \text{Mod}_\tau(\psi).$$

We write for this $L_1 \prec L_2$.

(ii) $L_1$ is implicitly reducible to $L_2$, if $T_1 \subseteq T_2$, for each $\tau \in T_1$, $\text{Str}_2(\tau) \subseteq \text{Str}_1(\tau)$, and for every implicit definition over $\tau$ via $\varphi \in \text{Fml}_1(\tau)$ with $\tau \subseteq \tau'$ there is an implicit definition over $\tau'$ via $\psi \in \text{Fml}_2(\tau')$ such that

$$\text{Mod}_\tau(\varphi) \cap \text{Str}_2(\tau') \downarrow \tau = \text{Mod}_\tau(\psi) \downarrow \tau.'$$

We write for this $L_1 \prec_{\text{impl}} L_2$.

(iii) We say that $L_1$ and $L_2$ are explicitly (implicitly) equivalent, if both $L_1 \prec L_2$ and $L_2 \prec L_1$ (if both $L_1 \prec_{\text{impl}} L_2$ and $L_2 \prec_{\text{impl}} L_1$).

Note that when both logics are $\Delta$-closed, the two notions are equivalent. Implicit reducibility, however, is a very crude measure. Most of the complexity features get lost, as theorem 17 in chapter 2 illustrates. On the other hand it really captures the implicit expressive power of a logic: two logics are implicitly equivalent whenever they are inherently related to each other, and in particular, they are not even implicitly equivalent, when they differ in a very deep sense. For instance, as we shall see in section 2.7, all Codd-complete query languages are implicitly equivalent, but by no means explicitly. As follows from [Makowsky 1980] and [Meyer-Parikh 1981], many dynamic logics are implicitly equivalent, but differ considerably on the explicit level. As we shall see in chapter 5, adding various forms of indeterminism and fairness to nondeterministic programs gives rise to logics which are not even implicitly equivalent to the usual dynamic logics [Francez-Katz-Makowsky 1982], but on the other hand, various forms of fairness [Lehmann-Pnueli-Stavi 1981] will turn out to be implicitly equivalent. Our thesis here is that explicit reducibility is a quantitative measure, whereas implicit reducibility is a qualitative measure. The former can obviously be even further refined by adding uniformity conditions to hold for the translations. A detailed discussion will be found in chapter 4.
2. Data Base Theory for the Relational Model.

2.1 Introduction

One of the most frequent application of computers nowadays is in data bases. There are various ways of modeling data bases, such as the network model or the hierarchical model, but the most widely studied for theoretical purposes is the relational model. An excellent reference is the textbook [Ullman 1982]. The entity-relationship model, cf. [Chen 1976, 1981] has not yet been really studied from a theoretical point of view. As much as I understand it, most of the theoretical results for the relational model carry over to the entity-relationship model, such as computable queries and dependency theory, but, to the best of my knowledge, no serious attempt has been undertaken to carry out such a task.

In what follows we concentrate on the relational model. It consists of families of data base states, which are divided in acceptable or consistent states and inconsistent states. Those are distinguished by constraints or dependencies. The consistent states are the models of the dependencies. With data base states we can do two things: We can ask queries or we can perform transactions. To complicate matters this is usually done by many users at the same time; we speak therefore of concurrent users. Transactions map consistent data base states into consistent data base states. They are usually decomposed into smaller operations which map consistent data base states sometimes into inconsistent data base states. There are two kinds of simple transactions: read only and write only. More complex transactions can be built from simple transactions by composition. Needless to say, all these operations should be computable. To sort out this mess a theory of transactions and concurrency control is in the making. The state of the art is described in [Date 1982], [Casanova 1981] and in the forthcoming book [Maier 1983]. An excellent survey is [Bernstein-Goodman 1982]. In this chapter we are only concerned with a special case of read only transactions, queries and dependencies. Neither general transactions nor concurrency control play a direct role. Indirectly, however, they serve as a motivation in our presentation of dependency theory.

Queries map data base states into relations. In [Chandra-Harel 1980] an abstract definition is given, the computable queries, which is the basis of our presentation in this chapter.

Data base states are structures like for first order logic, but for practical purposes some restriction are necessary. First of all, the structures are finite. Second, we distinguish between the relations representing the tuples in the data bases and the aggregate functions such as arithmetic operations or linear order on the entries. And third, we are not really interested in the underlying universes but only in the relations as such. In this chapter we shall not talk about the aggregate functions at all. To ensure that it makes no difference whether we talk about relations or structures we
introduce an invariance condition, called safety, which is discussed in section 2.2.

Dependencies are classes of data base states, usually the models of some first order sentences. They are grouped and classified according to syntactic criteria: Functional dependencies (FD), full implicational dependencies (FID), embedded implicational dependencies (EID), template dependencies (TD), multi valued dependencies (MVD) and embedded multi-valued dependencies (EMVD) can be conveniently described as classes of first order sentences with specific syntactic restrictions. In the sense of chapter 1, dependencies usually form a logic, with the vocabularies ranging over sets of relation symbols only and all the structures being finite. Queries are definable relations in this logic, and we shall see in section 2.6. that the implicit definition play an important role here.

Preservation theorems in logic are theorems which characterize classes of first order formulas having some semantic properties by showing that those sentences are exactly the ones which allow a special syntactic normal form. They are special cases of presentation theorems in the sense of chapter 1. But in contrast to universal normal form theorems (such as every first order formula is equivalent to a prenex formula) which usually are constructive, the normal form theorems coming from preservation theorems are often non-constructive. What we get is the following: The set of sentences $S$ having some semantic property $P$ is not recursive, but there is a recursive set $S_0$ such that every sentence $\sigma \in S$ is equivalent (over some first order theory) to a sentence $\sigma_0 \in S_0$. Though the theorem is non-constructive this has two advantages:

(i) We can, with no loss of generality, restrict ourselves - or for that matter the programmer of a data base system - to dependencies of the form $S_0$, and

(ii) by doing so, we know that property $P$ is a priori ensured.

If the property $P$ is one which is of intrinsic importance to our database system, then the restriction to sentences from $S_0$ will free the programmer from the correctness proof - or rather - force him to choose his dependencies carefully and prove then correctness before he is allowed to write them down.

Now in logic, the choice of the semantic properties $P$ is usually given in a natural way, say from algebraic considerations, and the problem is to find $S_0$. In data base theory the situation is reversed: We are given various candidates $S_0$, as the FD, FID, EID, MVD, TD, EMVD etc, and the problem we pose, is to define the corresponding properties $P$ which both characterize $S_0$ and are genuinely motivated by data base considerations.

It is our firm belief, that the syntactic restrictions given to various classes of dependencies are only meaningful iff they correspond to a semantic property which reflects data base practice. And it is such a property which should be called the meaning of a syntactic restriction. What we show here is giving meaning to being safe, typed and being a typed implicational dependency. What we propose furthermore is a program which consists of searching for the meaning of various other syntactic definitions of
dependencies such as template, embedded etc. Fagin went a good way to do this for typed embedded implicational dependencies [Fagin 1982] by showing that they are faithful (i.e. true in a product of finite non-empty relations iff true in each factor) and my previous [Makowsky 1981] proposed several such characterization, but their relevance for data base practice was not yet satisfactorially shown. In a forthcoming paper [Makowsky-Vardi 1983] more such results are collected.

Our main results here are:
(i) The complete characterization of equality generating dependencies based on separable dependencies wth have the subrelation property and are preserved under products.
(ii) The complete characterization of full typed tuple generating dependencies based on separable dependencies with the intersection property and the duplicate extension property.

The intersection property had been previously characterized in [Maier-Mendelzon-Sagiv 1979] as the property which guaranties the uniqueness of the completion operation in connection with the chase. Separability, however, is introduced here to give meaning to the restriction to typed formulas. It is discussed in detail in section 3 and captures the idea of separation of sorts, or attributes.

This chapter is organized as follows:
In section 2.2 we discuss a well known example from the above point of view, the definite formulas from [Kuhns 1969] and their syntactic characterization as permissible formulas, as described by many authors, e.g. [Cooper 1980], or as safe formulas, as described in Ullman's book [Ullman 1982]. We also note that the definite formulas are not recursively enumerable, as was shown by [Di Paola 1969].
In section 2.3 we follow the same pattern to propose a semantic characterization of typed, formulas. We also show that this class is not recursively enumerable. The results in this section are drawn from [Makowsky-Vardi 1983].
In section 2.4 we discuss FID's and FD's and connect the typed FID's (TFID) to the intersection property of relations. The results here are continuations of our previous work [Makowsky-Vardi 1983]. We end our presentation with some final remarks and open problems.
In section 2.5 we discuss decidability and complexity results for the consequence problem for various classes of dependencies.
In section 2.6 we give a brief presentation of the theory of computable queries and in section 2.7 we draw some conclusions and present some more open problems.
2.2. Safety (definiteness, domain independence).

Already in 1967 [Kuhns 1967] it was realized that first order formulas, which are relevant for database dependencies or queries, should satisfy an invariance condition. Intuitively this condition says that it does not matter if we speak of a relation or of a first order structure containing this relation.

Definitions: Let $R$ be a finite relation on $\prod A_i$ and let $A = \langle A_1, A_2, \ldots, A_n, R \rangle$ be the corresponding relational structure. Let $A^*$ be the relational structure obtained from $A$ by addition of exactly one element $a_i$ to every sort $A_i$ and not extending $R$. Kuhns calls the formulas $\sigma$ which are true in $A$ iff they are true in $A^*$ definite. A class $\Gamma$ of database states is called definite if it is closed under the formation of $A^*$.

Fagin [Fagin 1982] independently looked at this property and called it domain independence. Let $S$ denote the class of definite first order formulas, and $S^k$ be the class of definite formulas with at most $k \geq 0$ free variables. Di Paola showed:

Theorem 1: $S^k$ is not recursively enumerable for any $k \geq 0$.

For a proof one may also consult [Vardi 1981]. Note that if we allow infinite relations, we only get that $S^k$ is not recursive.

Here we have a non-recursive set of formulas $S$ and we would like to find a recursive set $S_0$ such that every formula $\sigma \in S$ is equivalent to a formula $\sigma_0 \in S_0$.

Let $S_0$ be the set of safe formulas as in Ullman's book [Ullman 1982] (or equivalently the set of permissible formulas from [Cooper 1980]). If we allow infinite relations, it follows easily from results in model theory that every formula of $S$ is equivalent to a formula in $S_0$. In fact we have even more:

Theorem 2: Let $\Sigma$ be a first order theory. Call a formula $\Sigma$-definite if it is definite on the class of finite models of $\Sigma$. Then the following are equivalent:

(i) $\sigma$ is $\Sigma$-definite and
(ii) In all finite models of $\Sigma$ is $\sigma$ is equivalent to a formula in $S_0$.

For the proof we define an algorithm based on relativization, which maps arbitrary first order formulas into safe formulas and which preserves equivalence (for models of $\Sigma$) if and only if the original formula was safe. This does not contradict theorem 1, since it merely says that the set of first order formulas, on which this algorithm does preserve equivalence, is not recursive.

Formulas with free variables define relations. For first order formulas this gives us a special case of first order (explicitly) definable queries. The definition of definite is
naturally extended to this case. We will return to definite formulas in the section on
query languages.

2.3. Typed dependencies.

In this section we look at the class of typed and safe formulas, which we denote by $T_0$.
Clearly this a recursive set of first order formulas. We propose to define an operation
on finite relations, which intuitively corresponds to the introduction of different attributes (sorts) for the arguments of the relation.

Let $R \subseteq A^n$ be a finite n-ary relation over some domain $A$. Let $\pi_i(R)$ be the i-th projection of $R$ onto $A$, and $A_i = \pi_i(R) \times \{i\}$ We define a new relation $\overline{R}$ on $\prod_i A_i$ in the following
way: $((a_1,1),(a_2,2),\ldots,(a_n,n)) \in \overline{R}$ iff $(a_1,a_2,\ldots,a_n) \in R$.

We say that a first order formula $\sigma$ admits separation of attributes (sorts) or, shortly, is separable, if $\sigma$ is true about $R$ if it is true about $\overline{R}$. A class of data base states $\Gamma$ is
called separable if it is closed under the formation of $\overline{R}$.

Remarks:
(i) If all the $\pi_i(R)$ are disjoint then $R$ is isomorphic to $\overline{R}$.
(ii) Using (i) we see that separable formulas are definite. This is due to our definition
of $A_i$ which is a projection. Had we defined it just to be a new copy of $A$, the results
below had to be slightly modified.
(iii) Functional dependencies are separable.

Let $T$ denote the class of separable first order formulas, and $T^k$ be the class of separable formulas with at most $k \geq 0$ free variables. Using a similar argument as in [Vardi
1981] one gets:

**Theorem 3:** $T^k$ is not recursively enumerable for any $k \geq 0$.

**Problem:** Is the class of EID's which are in $T$ recursive?

That separable formulas really capture the separation of attributes (sorts) is shown in
the following theorem:

**Theorem 4:** Let $\Sigma$ be a first order theory. Call a formula $\Sigma$-separable if it is separable
on the class of finite models of $\Sigma$. Then the following are equivalent:
(i) $\sigma$ is $\Sigma$-separable and
(ii) $\Sigma$ proves that $\sigma$ is equivalent to a formula in $T_0$.

The proof is similar to the proof of theorem 2.
2.4. Implicational Dependencies.

We are now in a position to define more classes of dependencies:

Definitions:
(i) A first order formula over a set $\tau$ of relation symbols is a full implicational dependency (FID), if it is of the form

$$\forall \varepsilon \land b_i(\varepsilon) \rightarrow b(\varepsilon)$$

where each $b_i$ is an atomic formula not containing the equality symbol, $b$ is atomic possibly containing equality and each variable which occurs in $b$ also occurs in some $b_i$. Note that we do not allow the empty conjunction.

If $b$ is an equality we also speak of equality generating dependencies (EGD), and if $b$ is an instance of a relation symbol we speak of tuple generating dependencies (TGD). The functional dependencies (FD) are the EGD's with only two $b_i$'s.

(ii) The classes TFID of typed full implicational dependencies, typed tuple generating dependencies TTGD and typed equality generating dependencies TEGD are defined analoguously.

(iii) The class of embedded implicational dependencies EID, consists of first order formulas of the form

$$\forall \varepsilon \land b_i(\varepsilon) \rightarrow \exists y \land c_j(\varepsilon,y)$$

where the $b_i$'s are as for the FID and the $c_j$'s are atomic with all the variables from $\varepsilon$ occurring already in the $b_i$'s.

(iv) The class of embedded template dependencies ETD, consists of the EID's with only one formula $c_j$, which is not an equality. In contrast to some papers in the literature we allow EID's to be untyped. A special case of template dependencies are the inclusion dependencies IND, where there is also only one formula $b_i$.

(v) The classes TFID,TEID,TETD,TIND of typed embedded dependencies are defined similarly.

An important subclass of TID are the Functional Dependencies FD. Let $X$ be a set of first order formulas and $E(X)$ denote the set of first order formulas which are equivalent to some formula in $X$. The following is a useful observation:

**Proposition 5:** (Beeri and Vardi) Every typed full implicational dependency is equivalent to a conjunction of a TGD and a EGD.

As was observed by Vardi and the author we have

**Theorem 6:** Both $E(FID)$ and $E(TGD)$ are not recursive. Neither is $E(FD)$.

A proof, due to Vardi, may be found in [Makowsky 1981]. It could also be proved using methods similar to [McNulty 1979].
Under what conditions can we axiomatize classes $\Gamma$ of data base states by dependencies of prescribed syntactic form? Let us look first at TFID's. Clearly they are again definite and separable. They also are preserved under cartesian products (the \emph{product property}). Given $\sigma \in FD$ and a relation $R$ and a subrelation $R_0 \subseteq R$ then $\sigma$ is true about $R$ iff it is true about $R_0$. (This is not generally true for $FID$.) Let us call this last property the \emph{subrelation property}, both as a preservation property for formulas $\sigma$ as well as a closure property for classes of data base states $\Gamma$.

The subrelation property is very strong and dependencies which satisfy it are invariant under losing any portion of your data bases. Its integrity can not be destroyed by deleting data. Note that the subrelation property is stronger than the \emph{substructure property} in model theory, because here we really take subsets of the relation, whereas in model theory we take subsets of the domains and consider the relation naturally induced on them. The substructure property is true for $FID$. In fact we have \cite{Makowsky-Vardi 1983}:

\textbf{Theorem 7}: A class $\Gamma$ of data base states, closed under isomorphisms, is axiomatizable by a set of typed equality generating dependencies $TEGD$ iff $\Gamma$ is

(i) separable,
(ii) has the subrelation property and
(iii) contains the trivial structure and
(iv) is closed under products.

The \textit{trivial structure}, is the structure which has exactly one element of each sort and all the tuples satisfy all the relations.

Since the compactness theorem is not true, if we only consider finite structures, theorem 7 can not be stated, like theorem 2 and 4, for single formulas.

Similarly we can define the \textit{intersection property}, which requires that if $\sigma$ is true in two relations $R_1, R_2$ then it is also true in $R_1 \cap R_2$. Again we have two versions of it, one as a preservation property and the other as a closure property for classes of structures.

Clearly the subrelation property implies the intersection property, but the intersection property seems more natural: Not every subset of a library catalogue is necessarily a catalogue, but we definitely expect the intersection of two catalogues to be a catalogue.

The intersection property is true for $TGD$ and for $FD$ but not for $FID$.

\textbf{Proposition 8}: If a separable formula has the intersection property, then it has the substructure property (but not necessarily the subrelation property).

The proof is purely semantical and uses the fact that we can represent every substructure as the intersection of two relations by renaming.
A last such property we want to consider is preservation (closure) under duplicate extensions. This is like logic without equality, i.e. we allow multiple occurrence of elements. More precisely, let \( a \in A, b \notin A \) and \( h \) be a mapping such that it is the identity on \( A - \{a\} \) and \( h(a) = b \). We have a natural extension of \( h \) to \( R \). Now \( < A \cup \{b\}, R \cup h(R) > \) is a duplicate extension of \( < A, R > \). With this we have ([Makowsky-Vardi 1983]):

**Theorem 9**: A class \( \Gamma \) of data base states, closed under isomorphisms, is axiomatizable by a set of tuple generating dependencies \( TTGD \) iff \( \Gamma \) is
(i) definite
(ii) closed under duplicate extensions,
(iii) the intersection property and
(iv) contains the trivial structure.

For typed dependencies we have the following analogue to theorem 9, also from [Makowsky-Vardi 1983]. Similar theorems can also be stated for the other cases.

**Theorem 10**: Let \( \Sigma \) be a set of first order formulas such that
(i) \( \Sigma \) is true in the trivial structure,
(ii) is separable,
(iii) has the intersection property and
(iv) preserves duplicating extensions.

Then \( \Sigma \) is equivalent to a set of typed tuple generating dependencies \( TTGD \)'s.

All the properties above but the closure under products have natural justifications in terms of data base practice. We had previously characterized \( TFID \) in terms of the Armstrong property (cf. [Fagin 1982, Makowsky 1981]) and a strong form of the finite model property [Makowsky 1981], but theorem 10 seems more natural. The Armstrong property is version of the weak generic structures, as dealt with in chapter 3, adapted to database theory. The finite model property in question is related to the class of securable formulas as defined in section 2.5.

### 2.5. The Consequence Problem

For various classes of dependencies the consequence problem has been studied. In general it is stated as follows: Given a finite set \( \Sigma \) of dependencies in \( D \), and a single dependency \( \sigma \in D \), can we decide whether \( \sigma \) is true in all (finite) relations satisfying \( \Sigma \)? This is closely related to the consequence problems in logic, with the difference that here we are mainly interested in finite models over relation symbols only and that the class of formulas \( D \), is of very low quantifier rank. Additionally, if we look at typed dependencies, we can not use variables repeatedly in different positions. Though a logician would expect undecidability results, if the formulas involve both existential and...
universal quantifiers, there is still place for many decidable subcases. Clearly if the dependencies are boolean combinations of purely universal and purely existential formulas, the consequence problem is decidable ([Bernays-Schönfinkel 1928], see also [Lewis 1979, 1980]). This class of formulas has also been extensively studied in model theory (cf. [Tharp 1974] and [Makowsky 1975]), and they were called secureable or continuous formulas. They have many nice properties:

Proposition 11: Let $\mathcal{S}$ be the class of secureable formulas. Then
(i) $\mathcal{S}$ is closed under boolean operations and
(ii) The class of valid and of finitely valid formulas in $\mathcal{S}$ coincide.

Clearly we get from this that the consequence problem for secureable formulas is decidable. In fact, the exact complexity of this consequence problem is known. The reader not familiar with complexity classes should consult [Garey-Johnson 1979].

Theorem 12: ([Lewis 1980]) There are constants $c>d>1$ such that the consequence problem for secureable formulas without function symbols or equality can be solved in $\text{NTIME}(c^n)$ but not in $\text{NTIME}(d^n)$.

Secureable formulas have the finite model property and are closed under boolean operations. This leads us to the following problem:

Problem: Do the properties (i) and (ii) proposition 11 characterize $\mathcal{S}$ up to logical equivalence, i.e. given $\mathcal{S}$ satisfying (i) and (ii), is it true that every formula in $\mathcal{S}$ is equivalent to a formula in $\mathcal{S}$? If this is not the case, is there such a maximal class for (i) and (ii), or what properties have to be added to get maximality?

Note that by a folklore result in model theory [Shoenfield 1967, problem 10c, p.97] $\mathcal{S}$ is characterized by the fact that every formula $\varphi \in \mathcal{S}$ both $\varphi$ and $\neg \varphi$ are preserved under unions of chains. Some applications of secureable formulas in characterizing dependencies may be found in [Makowsky 1981], cf. also section 2.4.

The first undecidability result for a class of dependencies, which was introduced previously in data base theory, appeared in [Chandra-Lewis-Makowsky 1981]. There it is shown that the consequence problem over finite relations for typed embedded implicative dependencies $\text{TETD}^-$ is not even recursively enumerable. This has been later improved to the class of typed template dependencies $\text{TTD}$ by [Vardi 1982] and [Gurevich-Lewis 1982].

Theorem 13: ([Gurevich-Lewis 1983]) Let $\Sigma$ range over finite sets of $\text{TETD}$'s and $\sigma$ over elements of $\text{TETD}$. Then the following two sets are effectively inseparable:
(i) The pairs $(\Sigma, \sigma)$ such that $\Sigma \models \sigma$ ( $\Sigma \models \text{find} \sigma$)
(ii) The pairs $(\Sigma, \sigma)$ such that $\sigma$ fails in some finite data base which satisfies $\Sigma$. 
It follows immediately that neither set is recursive and that there is no recursive axiomatization for $\models_{\text{finite}}$ for $TETD$.

As we know from theorem 12, the consequence problem for $FID$ is decidable in exponential time. The most successful algorithm for this is the $CHASE$ introduced in [Maier-Mendelson-Sagiv 1979]. Some of its limitations are discussed in [Goodman-Shmueli 1981]. Its popularity derives from the fact that it runs rather fast on interesting subclasses of $FID$. An abundance of complexity results for such cases may be found in [Maier-Sagiv-Yannakakis 1981]. However, for the general case of typed and untyped $FID$'s we have:

**Theorem 14:** ([Chandra-Lewis-Makowsky 1981])

(i) The consequence problem for $TFID$ can be solved in $\text{DTIME}(c^{n/\log n})$ but not in $\text{DTIME}(d^{\sqrt{n/\log n}})$ for some constants $c,d>1$.

(ii) The consequence problem for $FID$ can be solved in $\text{DTIME}(c^n)$ but not in $\text{DTIME}(d^n)$ for some constants $c>d>1$.

These results apply in particular to the $CHASE$ algorithm. It is sometimes argued, e.g. by V. Pratt, that simply exponential algorithms are suitable for computers. If we accept this we may argue that the $FID$'s are the largest class of dependencies which are reasonable for data bases. Further evidence for this view stems from the fact that they were independently proposed in various disguised forms, e.g. in [Papadimitriou-Yannakakis 1982], [Fagin 1982], [Beeri-Vardi 1981] or [Paredaens 1982]. The first three papers also introduce the $EID$'s. A good source for the history is [Fagin 1982].

The question which remains open till today, is whether every reasonable subclass of embedded dependencies has an undecidable consequence problem. The hard case seems to be the following:

**Definition:** A dependency is a *embedded multivalued dependency* if it is of the form

$$\forall a,b,c_1,c_2,d_1,d_2((P(a,b_1,\ldots,b_n) \land P(a,b_2,\ldots,b_n)) \Rightarrow d_2 P(a,b_1,c_1,d_3)).$$

Here all the $a,b,c,d$'s are vectors of variables.

We do not enter here the discussion of the importance of multivalued dependencies. The reader is referred to [Fagin 1977] or [Ullman 1982]. The reason we introduce them here is the following:

**Problem:** Is the consequence problem for embedded multivalued dependencies decidable?

To show how delicate such problems can be, let us look at the case of inclusion dependencies $IND$, which are a very special case of $EID$'s where both the hypothesis and the conclusion have length one and no equality is allowed. The precise definition was
stated in section 2.4. The following resumes what is known on the consequence problem for IND alone and for IND \cup FD.

**Theorem 15:**
(i) [Casanova et al. 1982]: The consequence problem for IND is \textit{PSPACE}-complete.
(ii) (Mitchell 1983 and Chandra-Vardi 1983, personal communication): The consequence problem for IND \cup FD is undecidable.

This is an example where a subset of EID which is not in FID has a decidable consequence problem, but where a very small extension leads to an undecidable consequence problem.

### 2.6. Query Languages.

A query \( q \) of type \( \tau \), where \( \tau = \tau_0 \cup \{ R_n \} \) is the similarity type of a class of data base states augmented by a new \( n \)-ary relation symbol, is a function which maps states(\( \tau_0 \)) into \( n \)-ary relations on the domain of these states. Since both states and relations are finite objects, we can code them in arithmetic and it makes sense to require that

(i) this function \( q \) is a \textit{partial recursive} function on these codes, i.e. there exists a Turing machine \( TM_q \), which computes the query on the codes.

On the other hand we do not want this function to be dependent on the particular codes, so we require also that

(ii) if two data base states \( D_1, D_2 \) are isomorphic, i.e. \( D_1 \cong D_2 \), so \( q(D_1) \cong q(D_2) \).

In [Chandra-Harel 1980] queries satisfying condition (i) and (ii) are called \textit{computable queries}. They argue convincingly that every reasonable query should be computable and say that a query language \( Q \) is \textit{complete} if every query in \( Q \) is computable and for every function satisfying (i) and (ii) there is a term \( q \in Q \) representing it. They also construct a complete query language \( QL \) which is based on relations only, i.e. without aggregate functions, but which leaves the arity of the relation symbol \( R_n \) dynamic. \( QL \) also has \textit{while}-statement built in.

Query languages actually used in practice or studied in the literature are not complete. However, two such languages SQL and QBE can be completed, without violating their main design principles. In [Makowsky-Zvieli 1983] we show how to complete those two languages with static arity of \( R \) and by only adding a recursive \textit{insert}-procedure, rather than a \textit{while}-statement.

The basic idea behind this comes from realizing that there is an alternative, more model theoretic, definition of the computable queries. Let us look at the graph of the above function \( q \). It is given by a class of structures of type \( \tau \) such that

(iii) for every finite \( \tau_0 \)-structure \( A \) there is at most one, up to isomorphism, relation
<A,R^d> in the graph of q.
We realize that this is exactly a statement of the form
"The graph of q defines R implicitly",
as we know it from model theory. Now there are various forms of implicit definitions,
depending on the use of additional predicates or even extensions of the domains. However,
if we allow unrestricted extensions of the domains, even the first order implicitly
defined relations on finite structures are not in general computable. On the other
hand, without additional predicates and extensions of the domains, only a restricted
class of computable queries can be obtained. In fact, it follows from [Fagin 1974] that
the queries which arise from first order implicit definitions without extensions of the
domains are exactly the NP-recognizable classes of finite structures which are closed
under isomorphisms. To get things under control we introduce a notion of implicit
definition with recursively bounded extensions of the universe, which we parametrize
by families of recursive functions. With the help of these concepts we ([Makowsky-
Zvieli 1983]) can show that

**Theorem 16:** The computable queries are exactly the first order recursively bounded
implicitly definable queries.

This theorem can be viewed as another presentation theorem in the sense of chapter
1. It is also another illustration of Kreisel's program, since it says, that on finite struc-
tures the computable queries are &Delta;_closed for recursively bounded implicitly definable
queries. In other words, it exhibits the connection between definability and computa-
bility, which deepens the justification for the approach in [Chandra-Harel 1980]. The
language QL turns out to be an analogue of the recursive infinite extension of first
order logic, which also plays the role as the unifying logic for various versions of
dynamic and algorithmic logics, as discussed in chapter 5. The theorem also says that
all the Codd-complete, i.e. containing all explicit definable queries, and computable
query languages are implicitly equivalent.

To measure how they differ explicitly, one can now classify query languages according
to the complexity of the evaluation of its queries. This leads to various hierarchies,
with the complete languages on the top and the first order explicitly definable queries
on the bottom. Query languages which are in this hierarchy are called Codd-complete.
In [Chandra-Harel 1982] and [Vardi 1982] this hierarchy is investigated along tradi-
tional complexity measures, which are not obviously connected to the way those
queries are expressed. In [Immerman 1982] a different approach to complexity is sug-
gested, which is based on the complexity of the definitions of the queries. Theorem 15
is useful especially for this latter approach. However, from a practical point of view,
implicitly definable queries have one serious drawback:

**Proposition 17:** It is undecidable, given a first order formula \( \varphi \) over a vocabulary \( \tau \)
containing a predicate letter \( R \), whether \( \varphi(R) \) represents an implicit definition of \( R \).
In first order logic over arbitrary models Beth's Definability theorem tells us that every implicit definition is equivalent to an explicit definition. Over finite models this is false. To illustrate the power of implicit definitions we ([Makowsky-Zvieli 1983]) have an interesting application of an old result of [Friedman 1976].

**Theorem 18:** For every recursive function \( f \) there exist an explicitly definable query \( q \) such that

(i) there is an implicit definition \( \phi \) of \( q \) but every explicit definition \( \psi \) of \( q \) has length \( l(\psi) > f(l(\phi)) \).

In this context we can return to our theme of preservation theorems. Like for dependencies, we can look at queries, whose implicit definitions are given by formulas with certain syntactic restrictions. For instance, if the implicit definition is given by a formula from \( \text{FID} \), treating the relation variable like an ordinary relation symbol, we call them the *full implicational implicit queries* (\( \text{FIIQ} \)).

**Proposition 19:** Every query \( q \in \text{FIIQ} \) is an explicitly definable query.

The queries in \( \text{FIIQ} \) are queries written in the language of programming logic, as exemplified by the language PROLOG, with the additional property that they have a unique solution. In programming with PROLOG one is not interested in this case, but rather in the least fixed point. Proposition 18 justifies this point of view. In [Chandra-Harel 1982] the complexity of PROLOG queries is investigated: Though they do not form a complete query language, they go far beyond the explicitly definable queries.

**Problem:** Show that, for the explicit \( \text{FIIQ} \) from theorem 19, there is an exponential lower bound. Show also, that in theorem 18 we could restrict the implicit definition to be an embedded implicational implicit query \( \text{EIIQ} \). In other words, like for the consequence problem, the difference between embedded and full consists in not recursively bounded versus exponential lower bound.

### 2.7. Conclusions and some open problems.

What is the meaning of a syntactic restriction on first order dependencies, we asked in the beginning of this section. In cases like prenex normal forms the meaning is that one can always do it. In the case of safe formulas the answer is definiteness, as it was observed by many authors before. In the case of typed formulas the meaning is separation of attributes (separability), as we showed in section 2.3, in the case of equality generating dependencies the meaning is separability, the subrelation property and closure under product, and in the case of full typed tuple generating dependencies (\( \text{TTGD} \)) the meaning is separability, the duplicate extension and the intersection property. In the latter case we could also answer separability, the
substructure property and faithfulness, but this can only satisfy an algebraist, and is not relevant for data base theory. The intersection property is not only conceptually more appealing, but is also justified by its usefulness in connection with the "chase", as pointed out in [Maier-Mendelzon-Sagiv 1979, lemma 7].

In each of the above cases the meaning of the syntactic restriction was exhibited by proving an appropriate preservation theorem. The tools to prove such theorems are taken from model theory or at least derive very strongly from proof techniques well known to model theorists. But the properties which are preserved are directly derived from data base practice.

In each of the above cases the syntactic restriction also gives us a recursive set of formulas (dependencies) whereas the set of formulas equivalent to a restricted formula is not even recursively enumerable.

It is now natural to ask whether such characterization can also be given to other classes of dependencies. In [Makowsky 1981] we proposed such characterizations, but they are not satisfactory enough.

Problem: Give characterizations for embedded multi-valued dependencies, typed embedded implicational dependencies and template dependencies.

We have not yet studied the way normalization theory gives rise to syntactically defined dependency classes. But it seems natural to guess that there are preservation theorems stemming from normal forms of data base schemes. A step in this direction may be found in [Ginsburg-Hull 1982] and [Ginsburg-Spanier 1982]. Another direction along these lines are characterizations which describe more closely the structure of the vocabulary (similarity type). Being typed is such a property, but more relevant and fruitful is the distinction between cyclic and acyclic data bases, a topic which we unfortunately could not cover here, cf. [Goodman-Shmueli 1982]. A good survey is [Fagin 1983]. The distinction between those two types of data bases is also reflected in various complexity results. For an excellent survey of normal forms (in contrast to syntactic normal forms), cf. [Beeri-Bernstein-Goodman 1978], [Bernstein-Goodman 1980] and [Ullman 1982].

Queries were defined as computable functions of data base states which preserve isomorphisms. In the spirit of our approach it is natural to ask if there other invariance properties which should be considered. In [Cooper 1980] it is suggested that definite formulas play also a role in query languages. His point is that, whereas queries are in general only partial recursive function, queries explicitly defined by definite formulas lead to total functions. This line of thought deserves more attention. The way one could make use of separable queries is obvious: but not very promising. A modest step for queries definable by FID's was done in proposition 18. A related approach, though
different in spirit, may be found in [Goodman-Shmuel 1982], where aspects of query processing are used for syntactic characterizations. But what we have is rather a

Problem: Which invariance properties for queries give meaning to interesting syntactically defined classes of computable queries. And which of these classes reflect also some complexity issue?

The importance of the work in [Chandra-Harel 1980] lies not only in the clarity of their concepts. The complete query language $QL$ they propose is the decisive step to provide data base theory with a notion of operational semantics for all reasonable query languages. The completeness of $QL$ enables us to provide all other query languages which precise definitions of their semantics, by interpreting them in $QL$. The usefulness of $QL$ is only theoretical. But it also lead in [Makowsky-Tiweli 1983] to extensions of existing query languages such as SQL and QBE, cf. [Ullman 1982] for their definition. Those extensions turned out to be complete in the sense of [Chandra-Harel 1980].

The success of the approach of [Chandra-Harel 1980] suggests some further research: One can try to mimick these definitions in the context of the entity-relationship model for data bases, as proposed in [Chen 1978]. This would lead to a precise comparison of the power of the two models and probably to a precise notion of their equivalence. On the other hand one can try to generalize the concept of a computable query to other transactions and develop a theory of computable transactions. A transaction will also be a computable function mapping data base states into data base states, but it is less clear what kind of isomorphisms or other properties it should preserve. We suggest that the methods we have illustrated in this section may lead to interesting developments in a general theory of computable transactions.

3.1. Introduction.

Data structures are structures, usually finite, sometimes potentially infinite. Their main purpose in programming stems from the need to organize algorithms transparently, saving space and time wherever appropriate. This is particularly important when we want to build polynomial, especially linear time algorithms. As a modest example for logicians let us pose the following problem: Given a set of propositional Horn formulas, we want to test satisfiability. It is easy to find an algorithm which uses $O(n^3)$ time, and working a bit harder, even one that uses $O(n \log n)$ time. But only a careful choice of the data structure will give a $O(n)$ algorithm, cf. [Itai-Makowsky 1982]. Logicians are usually not trained to express their ideas that way. Much more dramatic results of this type, involving deep mathematics, were obtained e.g. by R.Tarjan for graph theoretic algorithms, for which he was awarded the newly created Nevanlinna Prize 1982. Good introductions to combinatorial algorithms using various data structures are [Even 1979] and [Reingold-Nivergelt-Deo 1977]. The latter contains also an annotated bibliography.

Abstract data types are abstract structures like in algebra, category theory or model theory. They arise in attempts to axiomatize the properties of the data structures, which are needed to prove correctness of the so designed algorithms. The abstraction stems from the need to distinguish between the intended data type and its implementation. Especially in modular programming or in correctness proofs of programs one has to distinguish further between what is true in a data type and what follows from the assumptions which were made explicit. All other assumptions should be considered implementation dependent. From our remarks in the introductory chapter it should be clear, that we try to separate between correctness of a program in a particular implementation and correctness which is provable from explicit assumptions, and therefore enhances portability. Specification theory is the model theory of these explicit assumptions, or rather of what we allow to be such assumptions, since what follows from them is true in all its models. Usually such assumptions are expressed as equations, or, more generally, as universal Horn formulas, and strict universal Horn formulas (i.e. non-degenerate implications) cf. [Goguen-Burstall 1983]. But the model theory is modified to the extent that not all algebras, but only the initial algebras are considered, or at least, play a special role. We try to attack the problem here from the point of view of abstract model theory, as described in [Barwise-Feferman 1983].

The computer scientist's point of view of specification of abstract data types is widely discussed in the literature of the last decade, and many formalisms and semantical approaches have been proposed. A very useful didactic essay on the "software engineering viewpoint" is [Bjorner 1980].
Data types are generally considered to be many sorted structures or algebras, and methods from universal algebra, category theory and model theory have more or less successfully been applied to study the various questions concerning modularization techniques and expressive power (see for example [Goguen et al. 1978], [Burstall-Goguen 1980], [Wand 1978], [Kamin 1980,1983], [Bergstra et al. 1981], [Ehrig et al. 1980], [Ehrig et al.1982], [Ernst-Ogden 1980]). While most of the work in this area is intended to contribute to the design of specification techniques of languages, or studies of recursiveness in connection with specifications, little is known about the consequences of the implicit assumptions which underly the proposed semantical concepts.

Studying the implicit assumptions of algebraic specification theory is a problem which is conveniently expressible as a problem of abstract model theory: We have to axiomatize the universe of discourse when we want to prove something about "all possible approaches to algebraic specifications". The main difference here is, that we have to be much more careful about the choice of closure conditions imposed on the specifications. It is not a priori clear that we have closure under negation, disjunction or any sort of quantification. And in fact our results will show that such closure conditions are not appropriate.

Here we want to exhibit some of these assumptions common to all of the above mentioned approaches, and shows that, surprisingly, they essentially determine the character of the possible specification languages. Our results confirm the particular choice of specification languages in the literature in the following sense:

Each of the languages we discuss is complete for a set of implicit assumptions, in other words, each such language satisfies the assumptions, and any other language which satisfies these assumptions has no more expressive power. In other words, we have here another instance of a presentation theorem in the sense of chapter 1 But it is evident that, to complete Kreisel's program, we still have to go much further in isolating more definability criteria and in extending the framework to allow more general concepts of data structures, such as envisaged in [Goguen-Burstall 1983].

Our results rely heavily on the assumptions we made on the vocabularies (similarity types) of the data structures. Though they hold also for many-sorted relational structures with heterogeneous function and relation symbols, covering for instance all of the examples in [Broy-Wirsing 1983], we do not know, at present, if they can be extended to other similarity types, as allowed in [Bloom-Wright 1982].

The results in this chapter are inspired by Mal'cev's characterization of free classes [Mal'cev 1954] and extend [Mahr-Makowsky 1982]. They are essentially taken from [Mahr-Makowsky 1983], which is an adaption of this characterization to framework of specification of abstract data types.
The notions used in this chapter are standard in universal algebra and logic and can be found for example in [Monk 1976]. Explicitly we assume signatures, (vocabularies) to be of the form \( \tau = (S,C,F,R) \), so including sorts, sorted names for constants, functions and relation symbols. Finitary signatures, (finitary vocabularies), are those where function and relation symbols have finite arity. Structures (including relations) and algebras are defined in the usual way. Renaming \( r: \sigma \to \tau \) for signatures \( \sigma \) and \( \tau \) denotes the bijective assignment of \( \tau \) to \( \sigma \) which is compatible with the sorting of the components of \( \tau \) and \( \sigma \). Renaming carries over to structures, and we denote by \( A^{(r)} \), with respect to a renaming \( \tau \), the structure which is identical to \( A \) except that its universe, constants, functions and relations are renamed according to \( \tau \). Basic formulas consist of atomic and negated atomic formulas (including equations and inequalities) with free variables. Free variables could also be treated as uninterpreted constants. Basic sentences are variable free. For a structure \( A \), the set of basic sentences holding true in \( A \) is called the (basic) diagram of \( A \). If \( \Sigma \) is a set of formulas (possibly infinite) we denote by \( A, \Sigma \), \( \forall \Sigma \) the conjunction (resp. disjunction) of all the formulas in \( \Sigma \).

3.2. The Axiomatic Framework.

To prove statements about all possible specification languages we have to make precise what we mean by "all possible specification languages". In this section we give such a definition. The only objection it could provoke is being too general. But since our theorems hold for it, they will a fortiori hold for any narrower concept of specification languages, so we do not have to be bothered by this discussion here.

**Definition:** A semantical system is given by a pair \( (T,A) \) consisting of:
- a class \( T \) of finitary signatures and a family \( \Delta = (C_i)_{i \in I} \) of classes of type \( (i) \) structures
- where type \( : I \to T \) associates with each index \( i \in I \) a signature type \( (i) \) such that the following axioms hold:

1. **Isomorphism Axiom:** Given \( \tau \)-structures \( A,B \), and a class \( C_i \) from \( \Delta \). Then \( A \cong B \) implies that \( A \in C_i \iff B \in C_i \).

This axiom merely says that we deal with abstract data types, i.e. we are only interested in its isomorphism type and not in its particular representation. This is just the isomorphism axiom from abstract model theory, as defined in chapter 1.

2. **Renaming Axiom:** Given a renaming \( r: \tau \to \sigma \) and a class \( C_i \) with type \( (i) = \tau \), then there exists \( j \in I \) with type \( (j) = \sigma \) such that:

\( A \in C_i \iff A^{(r)} \in C_j \) for all \( \tau \)-structures \( A \).

This axiom just says that we can change names of relations or functions without affecting the structures. For example we can change from additive to multiplicative notation when dealing with a group without affecting the group itself. This is just the renaming axiom from chapter 1.
3. Intersection Axiom: For all indices $i,j \in I$ there is index $k \in I$ such that $C_k = C_i \cap C_j$.

This axiom ensures that the union of two specifications is again a specification. Note that here, in contrast to abstract model theory, we actually axiomatize the notion of "sets of sentences", rather than sentences. We get therefore conjunctions for free, but can avoid the other closure operations in our basic definitions.

4. Empty Class Axiom: For each $\tau \in T$ there is $i \in I$ such that $\text{type}(i) = \tau$ and $C_i = \emptyset$.

This axiom merely says that we can specify the empty class of $\tau$-structures.

The next step in our definition ensures that we are always allowed to add new constant symbols. We could be more liberal and also allow free use of new relation and function symbols, but our main results shows that this does not change anything.

Definition: A semantical system $(T,\Delta)$ is rich enough if additionally the following axiom (5) holds:

5. Richness Axiom: If $\tau = (S,C,F,R) \in T$, then for all families of constant symbols $C'$ over the same sorts $S$ such that $C \subseteq C'$ also $\tau' = (S,C',F,R) \in T$; and for all $\tau \in T$ and all sets $B$ of basic $\tau$-sentences $\text{Mod}(B) \in \Delta$.

In other words, we can extend a signature by arbitrary sets of constant symbols and every set of basic (variable free) sentences defines a specification.

In computer science, unlike in model theory, the intended models have to be countable, and preferably also reachable. Recall that a structure $A$ is reachable if every element in a $\tau$-structure $A$ is the interpretation of a term over $\tau$. (Clearly every structure can be made reachable by adding enough constant symbols.) Since we want to use model theoretic methods, we do not restrict ourselves to such models, but demand, that every specification which has a model, has also a reachable model. More precisely:

Definition: A semantical system $(T,\Delta)$ admits reachable structures if it is rich enough and additionally the following axiom:

6. Reachability Axiom: For all indices $i \in I$ there is a reachable structure $A \in C_i$.

Remark: If $(T,\Delta)$ is rich enough, then for any reachable $\tau$-structure $A$ with $\tau \in T$ there is an index $i \in I$ with $A \in C_i$. $C_i$ can be chosen to be the class of models of the basic diagram of $A$, i.e. the set of atomic and negated atomic (variable free) sentences true in $A$. 
Definitions:

(i) Given a class $C$ of $\tau$-structures. Then $C$ is basic compact if for all sets $B$ of basic $\tau$-sentences $C \cap \text{Mod}(B) \neq \emptyset$ iff for all finite $B_0 \subseteq B$ $C \cap \text{Mod}(B_0) \neq \emptyset$.

(ii) We call a semantical system $(T, A)$ basic compact (or of finite support), if for all $i \in I$ the class $C_i$ is basic compact.

Note, if $C$ is first order definable, then $C$ is basic compact.

Basic compactness just says that if a set of basic sentences makes a specification inconsistent, then there is already a finite subset of basic sentences which makes it inconsistent. Clearly, any system axiomatizable by finitary rules has this property.

Examples:

(1) Let $T_K$ be the class of signatures containing only function symbols and constant symbols and let $\Delta_K$ be all the equationally definable classes. This gives us a basic compact semantic system which admits reachable structures.

(2) Let $T_F$ be as above and $\Delta_F$ be the all the classes definable by finite sets of equations. This gives us a basic compact semantic system, but it is not rich enough.

(3) Let $T_E$ be as above and $\Delta_E$ be the quasi-varieties (i.e. classes definable by sets of finite first order Horn formulas, cf. [Monk 1976]). This again gives us a basic compact semantic system which admits reachable structures.

(4) Let $T$ be the class of all signatures and $\Delta_u$ be the classes definable by first order formulas. Then we get a basic compact semantic system which is rich enough but does not admit reachable structures. However, if we restrict ourselves to classes definable by universal formulas, then it does admit reachable structures. If we allow infinitary clauses (cf. section 3.2) then we destroy compactness, but still get a semantic system which admits reachable structures.

(5) If $T_R$ contains only relation symbols and $\Delta_{DB}$ consists of classes definable by full implicational data base dependencies ($FID$), as in chapter 2, then we get a basic compact semantical system, provided we allow infinite data base states. If we allow only finite data base states, as in chapter 2, compactness fails. However, in both cases the system admits reachable structures.

(6) We still get a semantical system which is rich enough if take $T$ as above and let $\Delta_{HL}$ be the classes definable by sets of statements expressing partial correctness of programs, i.e statements of Hoare Logic. For terminology cf. [Harel 1979]. But here we loose both compactness and the reachable structures.

(7) If $L = (T, \text{Str}, \text{Fml}, \models)$ is a logic, then the classes of the form $\text{Mod}_+(\emptyset)$ with $\emptyset \subseteq \text{Fml}(\tau)$ form a semantic system. If $L$ satifies additionally the Basic Axiom then the resulting semantic system is rich.
3.3. A complete specification Language for rich semantical systems admitting reachable structures.

In this section we show that the existence of reachable structures together with the axioms of rich semantical systems already determines fairly well, what kind of syntax is appropriate for specification languages. The reader, however, should be warned: the infinitary language we first present is not the ultimate specification language. Our first theorem just illustrates how little we need to get our first presentation theorem. If we add basic compactness (a model theoretic substitute for axiomatizability), as in theorem 2, we get the more familiar finitary logics.

Definition: The language of infinitary clauses is given by \( L_0 = (T_0, \text{Alg}, \text{Fml}_0, \models_0) \) with: 

- \( T_0 \) the class of all finitary signatures and \( \text{Alg}(\tau) \) the class of all \( \tau \)-structures.
- \( \text{Fml}_0(\tau) \) consists of all infinitary clauses over \( \tau \), i.e. formulas which are infinite disjunction of basic \( \tau \)-formulas, possibly with infinitely many free variables. Formally, if \( B \) is any set of basic \( \tau \)-formulas (possibly with free variables), then \( V B \) is a infinite clause. Finally, for \( \phi \in \text{Fml}_0(\tau) \), \( \models_0 \phi \) is defined by \( \models_0 \phi \) if for every \( V B \in \phi \) the universal closure of \( V B \) holds in \( A \).

We denote by \( L_f \) the system which we get from \( L_0 \) by restricting it to finite sets \( B \) of basic formulas, and call it the language of finite clauses. \( L_f \) is logically equivalent to the system given by sets of universal first order sentences (cf. Example (4)). Both \( L_0 \) and \( L_f \) are logics satisfying the Basic Axiom.

Our next two theorems show that the language of infinite (finite) clauses is universal for semantic systems which admit reachable structures (and are basic compact). More precisely:

**Theorem 1:** Let \( (T_0, \Delta_0) \) be the semantical system resulting from the logic \( L_0 \) of infinitary clauses, i.e. \( C \in \Delta_0 \) if there is \( \phi \in L(\text{type}(i)) \) with \( C_i = \text{Mod}(\phi) \). Then

(i) \( (T_0, \Delta_0) \) admits reachable structures;

(ii) If \( (T, \Delta) \) is a semantical system which admits reachable structures, then \( \Delta \) is a subfamily of \( \Delta_0 \), i.e. for all \( C \in \Delta \) we have \( C \in \Delta_0 \), and thus \( C \) is \( L_0 \)-definable.

**Theorem 2:** Let \( (T_0, \Delta_f) \) be the semantical system resulting from the logic \( L_0 \) of finite clauses, i.e. \( C \in \Delta_f \) if there is \( \phi \in L_f(\text{type}(i)) \) with \( C_i = \text{Mod}(\phi) \). Then

(i) \( (T_0, \Delta_f) \) admits reachable structures and is basic compact;

(ii) If \( (T, \Delta) \) is a semantical system which admits reachable structures and is basic compact, then \( \Delta \) is a subfamily of \( \Delta_f \), i.e. for all \( C \in \Delta \) we have \( C \in \Delta_f \), and thus \( C \) is \( L_f \)-definable.

**Proof:** The first theorem is proved using the method of diagrams and the second follows from the first using compactness. A reader with no background in model theory should consult [Monk 1976], or any other beginning text in model theory. A complete
proof may be found in [Mahr-Makowsky 1982].

3.4. Typical models and initial algebras.

The notion of a semantical system is meant to capture the semantics of a specification language and interprets a class \( C \) in the system as the semantics of a single specification. However, specification of abstract data types often attaches a single structure as semantics to a specification, like the initial algebra approach [Goguen et al.1978] or the final algebra approach [Wand 1978]. In both cases the single structures have a distinguished position in the "specific" class \( C \), which characterizes them uniquely up to isomorphism. Note, that one could also propose to vary the notion of "isomorphisms", i.e. pass to different categories, than just the category of \( \tau \)-structures. In [Wirsing-Broy 1980] it was proposed to require that the category of finitely generated structures form a complete lattice, to allow other than universal first order formulas in the specification language. Our theorems below illustrate, why such an approach must run into certain difficulties.

On the other side there more possible choices of structures which are unique in their class, so additional arguments should be put forward when one chooses initial or final algebras.

One such argument may be found in the notion of \emph{generic} algebras or, what this really amounts to, the concept of \emph{proof by example}. If we write down a specification \( \Sigma \) of a data structure in some formally defined specification language \( L \), the intended data structure should satisfy \( \Sigma \), but nothing else. However, this is not possible, since some other statements in \( L \) might be logical consequences of \( \Sigma \). So the best we can hope for is a structure (algebra) \( A \) which satisfies \( \Sigma \) together with all the consequences of \( \Sigma \), but whenever some statement \( \sigma \in L \) is not a consequence of \( \Sigma \) then it is false in \( A \). In algebra such a structure is called \emph{generic for} \( \Sigma \). In data base theory such structures are called \emph{Armstrong relations}, (cf.[Fagin 1982]). The usefulness of this concept is that truth in the generic structure (an example) is equivalent to being a logical consequence of \( \Sigma \), i.e. it formalizes the notion of proof by example. This idea has recently also been \emph{M} exploited for testing programs, cf [Rowland-Davis 1981].

What we try to argue for here, is that behind the notion of the \emph{initial algebra} lies a similar concept, and that the uniqueness of the initial algebra is just one of the many nice properties it has. The following notion of \emph{D-free structure} captures the intention behind these two approaches.

**Definition:** Given a class \( T \) of signatures, and let \( P_T \) and \( N_T \) denote the atomic, respectively negated atomic, \( \tau \)-sentences for \( \tau \in T \). Then a class \( D \subseteq P_T \cup N_T \) is called a \emph{preference system} for \( T \) if

1. \( D \) is consistent, i.e. \( \text{Mod}(D) \models \phi \).
(ii) D is maximal, i.e. any D with $D \subset D \subset P_T \cup N_T$ is inconsistent, in other words $\text{Mod}(D) = \emptyset$.
Note that if D contains free variables, then D is consistent if the existential closure of D has a model.

Example: Let A be a $\tau$-structure and D be the set of all basic $\tau$-sentences true in A (the diagram of A). Then D is a preference system. In fact, every preference system can be obtained in this way. To obtain a preference system D with free variables in this way we just look at the free variables as distinct new constant symbols (or generators) and take as A the reachable model described by D and the new constant symbols.
Preference systems may also be useful to handle parametrizations without mechanisms for "parameter passing", as was pointed out to me by J. Thatcher. cf. [Goguen et al. 1978] and [Thatcher et al. 1962].

Definitions:
(i) Given a class $T$ of signatures and a preference system D for $T$. Let C be a class of $\tau$-structures with $\tau \in T$ and $A \in C$, then A is $D$-typical in C if $A \models \sigma$ implies $C \models \sigma$ for all $\sigma \in D$. If D contains free variables, we mean by $A \models \sigma$ that the universal closure of $\Sigma$ holds in A and similarly for $C \models \sigma$.
$D$-typical structures are a weak form of generic structures, as far as basic, variable free sentences are concerned. If D contains free variables and $D \subset P_T$, $T$ has no relation symbols, then they are exactly the generic structures.
(ii) A is $D$-free in C if A is reachable and $D$-typical in C.
$D$-free structures combine the requirements of reachability and genericity, as far as they are compatible. For the usual definition of generic algebras, it may well be that there are no reachable generic structures, even if both separately exist. More on generic structures may be found in [Gratzer 1979, Appendix 4].

Examples:
(1) (initial) Let $T$ be arbitrary and $D = P_T$. Then A initial in C iff $A$ $D$-free in C.
(2) (final) Let $T$ be arbitrary and $D \subset N_T$. Then A final in C iff $A$ $D$-free in C.
(3) In general, if D is the diagram of some structure A then a $D$-free structure B in C is as different from A as C permits, i.e. for $\sigma \in D$ $B \models \sigma$ only if for all $B \in C$ $B \models \sigma$.
This is why we call D a preference system.

Facts:
(1) If $A, A'$ are $D$-free in C then $A \equiv A'$.
(2) Let $D$ be an arbitrary preference system for $T$ and $C$ the class of all $\tau$-structures for given $\tau \in T$, then A is $D$-free in C iff the restriction of $D$ to $\tau$ is exactly the diagram of $A$. (Recall that the diagram of $A$ is the set of all basic sentences holding in A)
**Definition:** A semantical system \((T, \Delta)\) admits \(D\)-free structures for a given preference system \(D\) for \(T\) if \((T, \Delta)\) is rich enough and additionally satisfies the following axiom:

6'. Preference Axiom:
For all indices \(i \in I\) there is a \(D\)-free structure in \(C_i\).

3.5. A complete language for semantic systems which admit initial semantics.

In this section we show that the existence of \(D\)-free structures determines even more, what kind of syntax is appropriate for specification languages. Again, it does not suggest any particularly useful syntax, but it shows how few assumptions allow us to get much information. If we assume, furthermore, basic compactness, then we shall get the expected finitary Horn clauses.

**Definition:** Given \(T=T_0\), the class of all finitary signatures, and \(D\) a preference system for \(T\). Then the logic of infinitary \(D\)-Horn clauses \(L^D_\infty=(T_0, \text{Alg}, \text{FmLD}, \models_{\infty})\) is defined like \(L_\infty\), except that for a set \(B\) of basic \(\tau\)-formulas \(V B \in L^D_\infty(\tau)\) iff there is at most one formula in \(B\) in \(D\).

We denote by \(L^D_F\) the set of finite Horn clauses.

Note that if \(B\) is a set of basic sentences with \(B \cap D=\emptyset\) and \(d \in D\) then \(-d=b_0\) is not in \(D\) and the clause \(V (B \cup \{d\})\) is equivalent to the infinitary formulas \(\wedge B \rightarrow b_0\), which is indeed an infinitary Horn formula.

**Theorem 3:** Let \((T_0, \Delta^D_\infty)\) be the semantical system resulting from the logic \(L^D_\infty\) for a given preference system \(D\), i.e. \(C_i \in \Delta^D_\infty\) iff there is \(\Phi \in \text{FmLD}(\text{type}(i))\) with \(C_i=\text{Mod}(\Phi)\) . Then

(i) If \(D=(P_T\) then \(L_0, \Delta^D_\infty\) admits \(D\)-free structures and

(ii) If \((T, \Delta^D)\) is a semantical system which admits \(D\)-free structures, then \(\Delta^D\) is a subfamily of \(\Delta^D_\infty\), i.e. for all \(C \in \Delta^D\) we have \(C \in \Delta^D_\infty\), and thus are \(L^D_F\)-definable.

**Remark:**
(i) of the theorem has an additional assumption, which we conjecture not to be necessary. However, in the case of basic compact semantic systems this additional assumption is not needed.

**Theorem 4:** Let \((T_f, \Delta^D)\) be the semantical system resulting from the logic \(L^D_F\) for a given preference system \(D\), i.e. \(C_i \in \Delta^D_F\) iff there is \(\Phi \in \text{FmLD}(\text{type}(i))\) with \(C_i=\text{Mod}(\Phi)\) . Then

(i) \(L_f, \Delta^D\) admits \(D\)-free structures and

(ii) If \((T, \Delta^D)\) is a semantical system which admits \(D\)-free structures and is basic compact, then \(\Delta^D\) is a subfamily of \(\Delta^D_F\), i.e. for all \(C \in \Delta^D\) we have \(C \in \Delta^D_F\), and thus are \(L^D_F\)-
definable.

**Proof:** Part (ii) in both theorems follows from a result due to [Cudnovskii 1988] which was independently rediscovered via methods of category theory in [Andreka-Nemeti 1975] and in [Banaschewski-Herrlich 1976]. Part (i) in the infinitary case with $D=P_\tau$ may also be found there. To prove part (i) for general $D$ one has to prove a lemma:

**Lemma 5:** Let $\Sigma$ be a set of finite $D$-Horn formulas and $\sigma_1, \sigma_2 \in D$. Then $\Sigma \cup \{\sigma_1, \sigma_2\}$ is consistent iff for each $i = 1,2$ $\Sigma \cup \{\sigma_i\}$ is consistent.

**Proof of lemma:** This follows from a close analysis of the resolution method to check satisfiability of sets of clauses, together with compactness. For more details on resolution we recommend [Robinson 1979].

### 3.6. Relevance for Specification of Abstract Data Types.

The completeness results in the previous sections talk about the limitations in defining classes of structures by specifications. These limitations are not determined by the properties of particular specification languages, but are caused mainly by the assumption of admitting reachable or $D$-free structures. That such assumptions are reasonable will be discussed below. What should be pointed out here is, that we have turned the traditional question of finding reasonable semantics for given syntactic approaches, upside down: We have first defined axiomatically how our semantics should look like by extracting some of the key ideas and intuitions behind the [Goguen et al. 1978]-approach and then we proved that this determines, up to logical equivalence, pretty well what kind of a syntax is well suited for specification of abstract data types. It remains open, how this approach can successfully be extended to the more general framework as envisaged in [Goguen-Burstall 1985]. But I am convinced that a careful implementation of "Kreisel's program" will lead to various clarifications in the relative chaos of too many practical proposals. And the experience gained from abstract model theory may help us in asking at least the right type of questions. We did not deal for instance with the problem of "hidden functions", as stated e.g. in [Thatcher et al. 1982]. But it may well be that our notion of $A$-closure and implicit definability from section 1.3 points in the right direction to clarify this concept. In section 2.6 we have already given an example of how to use implicit definitions.

Let us now look, retrospectively, at two special cases, initial and final structures, and discuss the semantic assumptions more closely.
Specification of Abstract Data Types with Initial Semantics.

The so-called algebraic approach to data type specification originates in the work of [Liskov-Zilles 1974], [Guttag 1975] and [Goguen et al. 1978] and considers specifications to be sets of equations or implicational equations (=strict universal Horn formulas). The definable classes are varieties or quasi-varieties of many-sorted algebras which contain, uniquely up to isomorphism, an initial algebra. Several attempts to extend this approach have been made, namely to use arbitrary first order formulas (including relation symbols), see [Carvalho et al. 1980] and [Wirsing-Broy 1980]. The last theorem shows that any extension beyond universal Horn clauses is unsafe in the sense that it does not guarantee the existence of initial structures. In the spirit of chapter 2 we could say that the meaning of universal Horn formulas is exactly given by the initial semantics. Since equivalence to a set of universal Horn sentences is generally undecidable (see section 2.4), a specification language which admits initial structures, and which allows a reasonable syntax analysis, therefore should be the language $L^P$ with $D=P$. This observation also applies to Requirement specifications as introduced in [Ehrig 1981]. There a set of requirements (in a typical case a set of first order sentences) is meant to precondition the data type to be specified, or to restrict the class of structures. That such a set of requirements allows initial structures is thus of great importance. A language for such requirements again is bounded in its expressive power by $L^P$ with $D=P$.

Specification of Abstract Data Types with Final Semantics.

As a reaction to [Goguen et al. 1978] final semantics is proposed in [Wand 1978] to determine by a specification not only a single data type, but also its possible implementations. Specification techniques for the so-called final semantics approach are not equally well developed. See, however, [Kamin 1980, 1983] and [Hornung-Raulefs 1981]. The possibilities of specifying a class of implementations for the, up to isomorphism, uniquely existing final structure are bounded by $L^P$ or $L^P$ with $D=NT$, in a sense just like above.

3.7. A Word on Other Applications.

Some of our results may have other applications as well. As it turns out the programming language PROLOG, (cf. [Clocksin-Mellish 1982]) gains popularity and is even considered by some as the language of the fifth generation of computers. In PROLOG, one can specify data types directly in Horn formulas, and our results show that this choice is appropriate. In [Chandra-Harel 1981] the connection between PROLOG and query languages, as we described them in the previous chapter, is studied. They show that PROLOG, taken as a query language, is not complete, and determine exactly its position in their complexity hierarchy for query languages.
In PROLOG, the data types specified by Horn clauses, are always realized as the initial structures, provided space considerations allow it. Not provable is equated with not true. Our notion of preference systems suggests interesting variations for that. I can think of applications in domains, which usually are captured by "non-monotonic logics", as suggested in [Artificial Intelligence 1980]. Here $D$ can be thought of as a description of an ideal world, or some default assumptions, and the $D$–typical models as best approximations to or least deviations from $D$. Specifications expressed in $D$-Horn formulas give the description of the real world. Our characterization of $D$-Horn formulas implies that such best approximations always exist, and that, if they always exist, the specifications have to be written in $D$-Horn formulas. We plan to explore this aspect of our work in the future.
References.

(Note that the references have no claim for either completeness nor historic accuracy concerning priorities. They merely reflect the authors accidental readings.)

Andreka, H. and Nemeti, L.; Generalisation of variety and quasivariety concepts to partial algebras through category theory, Dissertationes Mathematicae (Rozprawy Math.) 204 (1982)

Artificial Intelligence vol 13 (1980), Special Issue on Non-Monotonic Logic

Banachewski, B. and Herrlich, H.; Subcategories defined by implications, Houston Journal of Mathematics 2.2 (1976) pp.149-171


Barwise, J.; Admissible sets and structures, Springer Verlag 1975.


Chandra, Ashok and Harel, David; Computable queries for relational data bases, JCSS 21.2 (1980) pp 156-178

Chandra, Ashok and Harel, David; Horn clauses and the fixpoint query hierarchy, preprint 1982,11 p
Chandra, Ashok and Harel, David; Structure and complexity of relational queries, JCSS to appear, preprint 30p


Cooper, E.C.; On the expressive power of query languages for relational databases, TR-14-80, Harvard University 1980

Cudnovskii, G.V.; Some results in the theory of infinitely long expressions, Soviet Math. Dokl. 9 (1968) pp.556-559

Date, C.J.; An Introduction to Data Base Systems, vol.1, 3rd ed., Addison-Wesley 1981

Date, C.J.; An Introduction to Data Base Systems, vol.2, Addison-Wesley 1983.


Fagin, R.; Multivalued dependencies and a new normal form for relational data bases, ACM TODS 2.3 (1977) pp.262-278.

Makowsky


Ginsburg, Seymour and Hull, Richard; Characterizations for functional dependency and Boyce Codd normal form families, preprint Feb 1982, 65 p


Goad, C.; Automatic construction of special purpose programs, Report No. STAN-CS-82-897, Stanford University 1982.


Immerman, N.; Number of quantifiers is better than number of tape cells, JCSS 22.3 (1961) pp. 384-406.

Immerman, N.; Relational queries computable in polynomial time, proceedings of the 14th annual ACM Symposium of Computing, 1982, pp. 147-152.
Makowsky


Lindstrom, P.; On extensions of elementary logic, Theoria vol.35 (1969) pp.1-11


McNulty, G.F.; The decision problem for sentences preserved under relations between structures, abstract, JSL 44.4 (1979) p.452.


Maier, D., Mendelzon, A.O. and Sagiv, Y.: Testing implications of data base dependencies, ACM TODS, 4.4 (1979) pp 455-469


Mal’tsev, A.I.; Quasiprimitive Classes of Abstract Algebras, in the Metamathematics of
Manders, K.L. and Daley, R.F.; The Complexity of the validity problem for dynamic logic,
to appear in Inf. and Contr.
Meyer, A.R.; Ten thousand and one logics of programming, EATCS Bull. (January 1980)
pp. 11-20.
Meyer, A.R. and Tiuryn, J.; A note on equivalence among logics of programs, preprint MIT
Dec 1981
298.
Monk, J.D.; Mathematical Logic, Springer 1976
Paredaens, J.; A universal formalism to express decompositions, functional dependen-
cies and other constraints in a relational data base, Th. C. Sc. vol. 19.2 (1982) pp. 143-
160.
Robinson, J.A.; Logic: Form and Function, North Holland 1979
Rowland, J.H. and Davis, P.F.; On the use of transcendentals for program testing, JACM
Sadri, F. and Ullman, J.D.; Template dependencies: A large class of dependencies in rela-
372.
Shmueli, O.; The fundamental role of tree schemas in relational query processing, Har-
vard University TR-16-81, 180 p
Shoenfield, J.R.; Mathematical logic, Addison-Wesley 1967.
Statman, R.; Structural complexity of proofs, Ph.D. Department of philosophy, Stanford
University, 1974.
Tarski, A.; Some notions and methods on the borderline of algebra and metamathemat-
Thatcher, J.W., Wagner, E.G. and Wright, J.B.; Data type specification: Parametrization
Ullman, J.D.; Principles of Data Base Systems, second edition, Computer Science Press
1982
Ullman, J.D.; Principles of data base systems, Computer Science Press 1980
Vardi, M.Y.; On decompositions of relational databases, FOCS 1982
Vardi, M.Y.; The complexity of relational query languages, Proceedings of the 14th
Vardi, M.Y.; The decision problem for data base dependencies, IPL 12.5 (1981) pp. 251-
254
Vardi, M.Y.; The implication and finite implication problems for typed template

Wand, M.; Final Algebra Semantics and Data Type Extensions, Indiana TR85 (1978)