Report Number: WUCS-82-12

1982-07-01

Abstract Database System (ADS): A Data Model Based on Abstraction of Symbols

Takayuki D. Kimura, Will D. Gillett, and Jerome R. Cox Jr.

ABSTRACT DATABASE SYSTEM (ADS):

A DATA MODEL BASED ON ABSTRACTION OF SYMBOLS

Takayuki D. Kimura
Will D. Gillett
and
Jerome R. Cox, Jr.

WUCS-82-12

July 1982

Department of Computer Science

Washington University

St. Louis, Missouri 63130

(314) 889-6122

This research was supported in part by the Department of Health and Human Services under Grant HS-03792 from the National Center for Health Services Research and under Grant RR-00396 from the Division of Research Resources of the National Institutes of Health.

Abstract

Abstract Database System (ADS) is a data model in which the notions of symbol and abstraction play a fundamental role in the formal description and structuring of databases. The mechanism of abstraction in ADS is based on the abstraction operator of the lambda calculus. Important concepts in data modeling, such as entity set, entity type, property, attribute, relation, association, constraint, type checking and cardinality, are formally representable in the framework of abstractions on symbols. Thus, the number of primitive concepts in ADS is relatively small.

In the ADS language, there are three levels of symbol interpretation: a symbol may denote the symbol itself (when quoted), what the symbol denotes (when not quoted), and what is meant by what the symbol denotes (when not quoted and the evaluation function is applied). This capability of unstratified control of symbol interpretation allows the ADS language to be its own meta-language, and makes it possible to treat in a uniform manner, data, schema of data, schema of schema, and so on.

Associated with each name in ADS are two descriptions involving the denotation of the name: the intensional descriptor describes what the name can denote, and the extensional descriptor describes what the name does denote. The fundamental law of semantic consistency in ADS states that for each name in a database, the extension must be consistent with the intension.

The integration of both intension and extension into a single data model provides database designers with flexibility in modeling facts and concepts that reflect changing user perceptions. It also provides a mechanism for differentiating extensional raw data from intensional derived or virtual data. The same mechanism can be used for the translation of user views.

ADS is part of an effort to develop a design methodology for an enduring medical information system, an area where frequent changes in the conceptual schema are anticipated and multi-level abstraction is advantageous.

Categories and Subject Descriptors: D.3.1 [Programing Languages]: Formal Definitions and Theory - Semantics; H.2.1 [Database Management]: Logical Design - Data Models; I.2.4 [Artificial Intelligence]: Knowledge Representation Formalisms and Methods - Predicate Logic

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: abstraction, instantiation, intension, extension, semantic consistency, unquote

Acknowledgements

The authors wish to thank Dr. John M. Smith of CCA for his constructive comments and criticism on earlier versions of the manuscript. Our discussions with Dr. Jack Minker of Maryland about logical databases and deduction capabilities have helped us to make refinements to the ADS data model. Thanks are also due to Ken Wong, Andy Laine and Stuart Goldkind for very careful reading of the manuscript and for constructive comments. Pat Moore has kept our feet on the ground throughout the development of the model by experimenting with ADS concepts on her neonatology database. Bill Ball and other members of the Information Systems Group have been good listeners and have provided various forms of support.

1. Introduction

This paper discusses the design philosophy of a new data model Abstract Database System (ADS) and describes some of its unique features. The design of the ADS data model and language is a part of a broader research project whose goal is the development of a design methodology for composite medical information systems capable of dynamic system evolution in response to user needs. The design methodology will be tested by the implementation of a composite medical information system in an operational environment that combines clinical and research activities [9].

Our goal has been to develop a design methodology for a database system that can survive for many decades:

without fundamental changes in its interface with users,

without periods of unsatisfactory performance due to growth, and

without jeopardizing the information stored.

These objectives must be achieved despite changes in:

the reality to be modeled,

the user's perception of the structure of the world, and

the technology available for system implementation.

ADS evolved from a study of the invariants (or fundamental concepts) in a user's perception of medical information. An implementation of the ADS model is being used to test the adequacy of the ADS invariants and to identify performance enhancement hardware.

The data model design guidelines are:

1) Formality: The model should be unambiguous and have a precise description which can serve as the foundation for the development of an enduring database design methodology.

2) Parsimony: The model should have a minimal number of constructs. Not only because minimization is of theoretical interest, but it leads to a better fundamental understanding of the model and a simpler implementation.
3) Universality: The model should provide enough flexibility to represent different views of reality during the long lifetime of a database.
4) Implementability: There must be an efficient implementation of the model.

Note that the ease of use (the human engineering aspects) of the ADS language is not included in the above primary requirements because ADS is intended to be a primary model in the sense of [17]. It is understood that secondary or vernacular models (based on the primary model) will be designed for production purposes with proper considerations for human engineering factors.

A set of conceptual tools organized for representing a user's knowledge about reality is called a data model [34]. Different data models utilize different sets of conceptual tools. The relational data model [8] uses the concept of mathematical relation. The Entity-Relationship data model [4] uses the concepts of entity set, relationship set, and mapping among them. The functional data model in DAPLEX [31] uses the concepts of set and multi-valued function. A binary data model (e.g., in [1]) uses the concepts of category (set) and access function (binary relation). The semantic network data model in TAXIS [26] uses the concepts of class, property, and generalization (IS-A relationship). The logical data model in MRPPS [25] uses the calculus of many sorted logic [12].

The ADS data model provides abstraction on symbols as the basic conceptual tool. The abstraction operator in ADS is the abstraction (lambda) operator in logic (e.g., the lambda calculus of [5]). A lambda expression is used to specify a set as well as a function. Thus, other functional or set-theoretical data models can be represented within the framework of ADS.

Other concepts in ADS are related to different ways of using symbols: intensional vs. extensional, naming vs. describing, and quoted vs. unquoted. At the core of ADS lies the concept of a symbol and its usage. Most of the fundamental concepts in ADS are imported or derived from logic (or the logical study of semantics). The abstraction operator is one of them and the intension/extension distinction is derived from the sense/denotation dichotomy of Frege [13].

The main characteristics of ADS are as follows:

(1) Symbols: An ADS database is a collection of symbols (data) whose syntax and semantics are formally defined. The ADS database system enforces semantic consistency among the symbols in the database by rejecting any symbols that would result in an inconsistent database.

(2) Naming and Describing: Symbols are used to name real world entities and to describe the relationships among them. They are also used to name and describe other symbols and abstractions on symbols. The database can be thought of as a set of names, each name having an intensional descriptor and (possibly) an extensional descriptor.

(3) Intension and Extension: The intensional descriptor of a name describes what the name can denote, and the extensional descriptor describes what the name actually does denote. Names can be used to refer to entities in a possible (intensional) world as well as entities in the actual (extensional)

world. The consistency between the possible world and the actual world is maintained by enforcing the fundamental law of semantic consistency. One user's actual world may be another user's possible world. Derived data can be separated from but easily integrated with the raw data.

(4) Abstractions on Symbols: Users can name and describe generalizations and discriminations on symbols (and therefore indirectly on real world entities). They can be used to represent the concepts of set, function, relation, property and attribute. By nested applications of abstraction operators, arbitrary levels of abstraction can be represented with a parsimonious set of language primitives.

(5) Unstratified Control: A symbol can be used to denote itself (when quoted), its denotation (when not quoted), or its denotation's denotation (when evaluated). With this capability, the object database and the meta database can be integrated into one database. Data, schema of data, schema of schema, and so on can be treated uniformly.

While most of the logical concepts in ADS are well known among logicians and semanticists, they are not common in the database literature. These concepts are introduced in a tutorial manner in Sections 2 through 5. Section 6 gives an overview of ADS. Section 7 demonstrates simple capabilities using a school database, and Section 8 illustrates more advanced capabilities of ADS. Section 9 concludes with future directions.

2. Symbols

A database is a collection of symbols (data) representing a set of views of reality as held by each member of a user community. Users share knowledge and perceptions of reality by communicating their judgments about reality through

a depository (database) of symbolic representations of such judgments. A precise and unambiguous interpretation of data is required for sharing knowledge.

A symbol is a medium of communication, generated by the sender and recognized by the receiver. When the sender generates a symbol (and sends a symbol token), he associates some meaning with the symbol based on a rule of interpretation, and expects the receiver to interpret the symbol in the same way. We will not differentiate the logical concept of a symbol from the physical concept of a symbol token [21]. We consider a symbol to be a logical object.

2.1 Identity of Symbols

A symbol may be simple or complex. A character is a symbol. Words, expressions, and sentences (as in English) are also symbols. A simple symbol has no structure. A complex symbol consists of component symbols and a structure which associates them. Two complex symbols are identical when they have the same component symbols and structure. While there may exist some disagreement about the meaning of a symbol between the sender and the receiver, there must be agreement about the identification of symbols for precise and effective communication among database users.

2.2 Sense and Denotation

A precise specification of a rule of interpretation is the central issue in semantic database design. Analysis of different kinds of meaning that can be associated with a symbol contributes to the precision of such a specification. According to Frege [13] there are two kinds of meaning associated with each symbol: sense and denotation. For example, let us assume that Cox teaches

CS360, and is also the chairman of the CS department. The two symbols:

(1) the chairman of the CS department

(2) the instructor of CS360

denote the same person Cox. Thus, in one way, they have the same meaning, called the denotation. However, the two symbols are not synonymous (having the same meaning in all respects) because the following symbols (sentences) are not synonymous:

(3) The chairman of the CS department is the instructor of CS360

(4) The instructor of CS360 is the instructor of CS360.

While (3) can convey information, (4) cannot; i.e., (3) and (4) have different meanings. Sentence (3) may be true or false depending on reality, but (4) is always true. Any difference between the meanings of (3) and (4) must be due to a difference between the meanings of (1) and (2) because the rest of the sentences are identical. Frege associates a meaning other than denotation, with (1) and (2), which he calls sense.

According to Church [6]:

> The sense of an expression (symbol) is its linguistic meaning, the meaning which is known to any one familiar with the language and for which no knowledge of extralinguistic fact is required; the sense is what we have grasped when we are said to understand the expression.

When a symbol is a declarative sentence, we regard its denotation to be a truth-value and its sense to be a proposition. Thus, (3) and (4) have the same denotation, Truth, but have different senses, i.e., two different propositions. In general, the sense determines the denotation.

A symbol denotes its denotation and expresses its sense. When two symbols express the same sense, they are synonymous. When two symbols denote the same denotation, they are equivalent. We will represent synonymy by '$\equiv$'

and equivalence by ':=:'. For example, under most circumstances, we may assert,

"the chairman of the CS department"

≡ "the head of the CS department",

and

"the chairman of the CS department"

:=: "the instructor of CS360".

For declarative sentences, assuming John and Dave are students,

"John is a student" :=: "Dave is a student" :=: "Truth".

Note that synonymy and equivalence are relations on symbols, not on objects. If two symbols are synonymous, then they are always equivalent, but the converse is not necessarily true.

There is no universally accepted theory on the exact nature of sense. One possible characterization of sense is to assume that two symbols are synonymous if and only if they are identical, i.e., no two different symbols have the same sense. This view is taken by Perlis in [27]. We adopt the same view in ADS, except that the user can define two different symbols to be synonymous.

2.3 Intension and Extension

In the literature of logic and semantics, different terms are used to refer to sense and denotation. For denotation there are 'reference', 'designatum', and 'extension', and for sense there are 'connotation', 'concept' and 'intension'.

The terms 'intension' and 'extension' are also used in logic in a way that is closely related to, but different from, the sense/denotation distinction. According to Mill [24], the word 'white' denotes all white

things (a class of objects), and connotes the attribute whiteness; i.e., the extension of 'white' is a class of objects, and the intension is the property shared by all objects in the class and by only those objects. Similarly, we can say that symbol (1) denotes a person (extension) and connotes a property (intension) of the person, and that symbol (2) denotes the same person but connotes a different property. For example, the extension of a mathematical function is defined as a set of ordered pairs of an argument and its value, while the intension of a function is defined as the rule of association between its arguments and the corresponding values [7]. Similarly, a set can be defined extensionally by explicitly enumerating its members, or can be defined intensionally by stating the properties each member of the set must have.

In the database literature [25], the term 'extensional database' is used to refer to the set of relational tuples (elementary facts), and the term 'intensional database' is used to refer to the set of general laws (general facts) with which the individual tuples must be consistent. The following example is a small logical database in which the first order predicate calculus is used to specify the intensional database:

Elementary Facts (Extensional database)

FACULTY (person)

Cox
Gillett
Kimura

TEACHING (instructor, course)

Kimura	CS135
Gillett	CS236
Cox	CS360

General Facts (Intensional database)

$(\forall x)(\forall y)(\text{TEACHING}(x,y) \rightarrow \text{FACULTY}(x))$

: In any instance of teaching a course, the instructor is a faculty member.

$(\forall x)(\forall y)(\forall z)((\text{TEACHING}(x,z) \wedge \text{TEACHING}(y,z)) \rightarrow x = y)$

: Every course is taught by at most one instructor.

We will use the terms 'intension' and 'extension' in accordance with the above database usage.

2.4 Significance

Symbols in a database are expected to be meaningful. A symbol, occurring in a particular context, is called nonsignificant if it denotes nothing; i.e., it has no denotation. A symbol is nonsensical if it has no sense. For example, assuming that John is a student who is not taking CS360,

the grade of John in CS360

is nonsignificant in the present context but is not nonsensical. On the other hand, presuming that 'EE280' denotes a course

the grade of EE280 in CS360

is nonsensical and is nonsignificant in any direct (nonquoted) context.

In ADS, the predicate "x is significant" is represented by '/x/' as in:

"the grade of John in CS360" is significant

≡ /"the grade of John in CS360"/

:=: Falsehood (if John is not taking CS360)

and,

"the instructor of CS360" is significant

≡ /"the instructor of CS360"/

:=: Truth (if someone teaches CS360).

2.5 Use and Mention

When users communicate with each other through a symbolism which describes another symbolism (as we do in this paper, discussing the semantics of symbols in English), it is important to distinguish between a symbol itself and its denotation, and to differentiate object symbols, which denote non-symbolic

objects, from meta symbols, which denote object symbols. For example, consider the following English statements as data representing user knowledge:

(5) John is a student

(6) "John" is a first name.

In (5), the word 'John' is an object symbol used to denote a person who is a student. In (6), the word 'John' is mentioned as a symbol belonging to the English language [21]. The expression '"John"' denotes the symbol 'John', and is a meta symbol.

Note that in this paper, we differentiate between quotes in data (') and quotes in English (").

2.6 Unquote

In order to facilitate better control over levels of symbol interpretation (object, meta, meta meta, and so on), we will use a left square bracket ([) preceding and a right square bracket (]) following a symbol to represent what is denoted by what the enclosed symbol denotes (indirect denotation). Thus, a symbol inside a pair of square brackets will be evaluated to a symbol which, in turn, will be evaluated when the complex symbol containing the square brackets is subsequently evaluated. We call the square brackets the unquote operation because it corresponds to the inverse of the quoting operation. (The expressive power of a similar operation has been studied in [27] within the framework of the first order predicate calculus.)

When the symbol enclosed in brackets denotes a non-symbol, the bracketed symbol along with the brackets is considered to be nonsignificant, i.e., the symbol denotes nothing.

Consider for example the following complex symbols:

(7) "The most common name" denotes "John"

(8) [The most common name] is a student

(9) ["John"] is a student

(10) [John] is a student.

Statement (7) establishes a semantic relationship between two symbols; the first phrase 'the most common name' is a symbol that denotes another symbol 'John'. Statements (8) and (9) are equivalent to (5), because the bracketed expression in (8) '[The most common name]' and that in (9) '["John"]' denote the same object as the name 'John' denotes. The symbol '["John"]' denotes a person John because it is what is denoted by what the symbol '"John"' denotes, i.e., what is denoted by 'John'. However, statement (10) is nonsignificant, because the symbol 'John' denotes a person and not a symbol, and a person does not denote anything. Only a symbol has the ability to denote something. Thus, the expression including the brackets '[John]' is nonsignificant and nonsensical.

2.7 Control Symbols and Unstratified Control

When a complex symbol is evaluated, a control symbol specifies how the process of evaluation should proceed. Quote and unquote are control symbols. Control symbols do not denote objects in reality; they are intrinsic to the interpretation of symbols in the language.

The availability of quote and unquote gives the ADS language unstratified control [15] over the levels of interpretation. The same symbol can be used as an object symbol in one context and as a meta symbol in another. All natural languages have such capability, as do machine languages for von Neumann type machines where instructions and data are stored in the same

memory in the same format. The LISP programming language [22] also has such a capability. The data model of Abrial [1] contains the concept of unstratified control in order to achieve self-representability of the model.

2.8 Conditional Control

The conditional control symbols '→' and ';' select the next symbol to be evaluated and is similar to the conditional expression in LISP. For example,

(John has an advisor → the advisor of John; the department chairman)

is equivalent to 'the advisor of John' if 'John has an advisor' is true; otherwise it is equivalent to 'the department chairman'. In general, for arbitrary symbols a, b and c, the following equivalences (:=:) hold:

(a → b; c)	:=: b	if a :=: "Truth",
	:=: c	if a :=: "Falsehood",
	is nonsignificant	otherwise.

Similarly,

(a → b)	:=: b	if a :=: "Truth",
	is nonsignificant	otherwise.

Note that the meaning of the conditional control symbol '→' in ADS is different from that of the implication symbol in logic. The symbols a and b in the above definitions need not be symbols that denote logical values. Even if the symbol b (or c) is nonsignificant, (a → b; c) may be significant. These are the only control symbols in ADS for sequencing.

3. Abstractions on Symbols

Smith and Smith [32] were the first to propose that the concept of abstraction be included in a data model. Their notion of abstraction consists of three operations for constructing new objects from existing ones: generalization

(set union), aggregation (cartesian product), and classification (set formation). These three operations are valuable conceptual modeling tools.

However, our notion of abstraction is different than theirs. Their theory of abstraction is a theory of objects. Abstraction is independent of the symbolism used for representing it. Our theory is a theory of symbols. In our view, abstraction is possible only through symbolism.

When users represent their knowledge about reality in a symbolic form, the first level of abstraction is achieved. This level of abstraction lies outside the database realm. For example, the decision to represent a person John by the symbol 'John' cannot be represented in the database unless the database already has some other symbolic representation of the person John. On the other hand, the database can deal with abstractions on those symbols that contain the symbol 'John'; the abstractions represent knowledge about the person denoted by the symbol 'John'.

3.1 Generalization and Discrimination

In ADS, generalization and discrimination are two complementary abstraction mechanisms. The mechanism of generalization is the same as the abstraction operator or the lambda operator in symbolic logic [5]. It is used to make an abstraction of similar patterns into one general pattern by identifying the similarity among the patterns. The discrimination mechanism is the inverse lambda operator, which is related to the description operator or the (inverted) iota operator in symbolic logic [28]. It is used to make an abstraction of similar patterns into one discriminating pattern by identifying differences among them.

Consider the following statements:

(11) John is a student

(12) Dave is a student.

These two statements (symbols) have the same form as

(13) ___ is a student,

where the blank can be filled with a symbol denoting a person. This form interpreted as a generalization displays the similarity between (11) and (12), i.e., they share the symbol 'is a student'. The form interpreted as a discrimination displays the difference between (11) and (12), i.e., they differ in the part indicated by the blank.

In ADS, these two distinct interpretations of (13), generalization and discrimination, are represented by (14) and (15), respectively.

(14) ()x:Person)(x is a student)

(15) ()/x:Person)(x is a student).

The symbol 'x', called the abstraction variable, plays the role of a place holder corresponding to the blank in (13). The common symbol 'x is a student' is the abstraction body.

Symbol (14) is a generalization descriptor, (15) is a discrimination descriptor, and both are abstraction descriptors. 'x:Person' indicates that x must be of type Person (ref. 3.3). ADS abstractions are primarily used to represent sets, functions and types. For example, abstraction (13) may represent the set of persons who are students, the predicate function on persons for recognizing students, or the type Student, with appropriate interpretation rules.

3.2 Instantiation

The inverse of abstraction is called instantiation. If abstraction (14) is the result of generalization on (11) and (12), then the instantiation of (14) yields (11) when based on the symbol 'John' and (12) when based on the symbol 'Dave'. Similarly, if abstraction (15) is to represent the discrimination of (11) and (12), then instantiation of (15) yields one discriminant, 'John', when based on complex symbol (11), and another, 'Dave', when based on complex symbol (12).

3.2.1 Intensional Instantiation

In ADS, we differentiate between intensional instantiation and extensional instantiation based on the synonymy/equivalence distinction. An intensional instantiation is represented by the dot notation:

(16) "John" . (λx:Person)(x is a student)

(17) "John is a student" . (γx:Person)(x is a student).

Here the symbol specified by the expression preceding the dot in (16), 'John', is the basis of the instantiation. Similarly, 'John is a student' is the basis of instantiation (17). The result of (16) is 'John is a student'. The result of intensional instantiation (17) is 'John'.

Recall that two symbols are synonymous ($\equiv$) if they express the same sense. We define (16) to be synonymous with (11). Similarly, we define (17) to be synonymous with any symbol x of type Person such that 'x is a student' is synonymous with (11). The only symbol that satisfies this condition is 'John'. Thus,

"John".(λx:Person)(x is a student) $\equiv$ "John is a student"

and

"John is a student".(γx:Person)(x is a student) $\equiv$ "John".

It is possible to interpret an abstraction as a specification of a one-to-one partial transformation on symbols when it is used with an intensional instantiation. Thus, generalization (16) maps one symbol ('John') to another ('John is a student'), and discrimination (17) is the inverse mapping of (16). Under this interpretation, a basis is an argument to a function, an intensional instantiation is a function application, and discrimination is the inverse of generalization.

3.2.2 Extensional Instantiation

An extensional instantiation is represented by the lowered star notation as follows:

(18) "John" $_{*}$ (λx:Person)(x is a student)

(19) "John is a student" $_{*}$ ($\forall$x:Person)(x is a student).

Instantiation (18) is synonymous, by definition, with a symbol that is equivalent to (having the same denotation as) (11), and (19) is synonymous with a symbol x of type Person such that 'x is a student' is equivalent to (11). Since there are many symbols equivalent to (11), (e.g. (12)), (18) is synonymous with an indefinite symbol that is non-deterministically chosen from the set of equivalent symbols. Thus, the following hold:

"John" $_{*}$ (λx:Person)(x is a student) :=: "John is a student"
:=: "Dave is a student"
:=: "Truth".

Similarly,

"John is a student" $_{*}$ ($\forall$x:Person)(x is a student) :=: "John"
:=: "Dave"

"Truth" $_{*}$ ($\forall$x:Person)(x is a student) :=: "John"
:=: "Dave".

Here 'Truth' denotes the logical value, Truth, the denotation of all true sentences.

In summary, the following are the semantic definitions of intensional and extensional instantiations:

Let

a and b denote arbitrary symbols,

A(x) be an arbitrary abstraction body, and

A(a) be the result of substituting a for all free occurrences of x in A(x).

Then, $a \cdot (\lambda x)A(x) \equiv b$ iff $A(a) \equiv b$,

$a \cdot (\gamma x)A(x) \equiv b$ iff $A(b) \equiv a$,

$a \ {}_{*} \ (\lambda x)A(x) \equiv b$ iff $A(a) :=: b$,

and $a \ {}_{*} \ (\gamma x)A(x) \equiv b$ iff $A(b) :=: a$.

The process of obtaining the result of an instantiation of generalization requires (i) binding (associating) the abstraction variable with the basis, (ii) checking the basis for the proper type, and (iii) substituting the basis for all free occurrences of the variable in the abstraction body. If the basis of the instantiation is not of the required type, then the symbol expressing the instantiation has no sense, and is therefore nonsensical. Similarly, the process of obtaining the result of an instantiation of a discrimination requires (i) matching the abstraction body with the basis, (ii) binding the variable with the corresponding symbol (discriminant) in the basis, and (iii) checking whether the discriminant is of the proper type. If the discriminant is not of the required type, then the symbol expressing the instantiation is nonsensical.

3.3 Type

In order to maintain the semantic integrity of a database, it is important that the database be free of nonsensical data and nonsignificant symbols. We define type as a linguistic mechanism that increases semantic data integrity by rejecting syntactically correct but semantically meaningless symbols.

Type checking is used to ensure the consistency between an abstraction and its instantiations. For example, when (11) and (12) are abstracted (generalized) into (14), it is intended to be a generalization of more than just (11) and (12). The intended extent of generalization is specified by the type Person associated with the abstraction variable 'x', which specifies the range of the variable. Any instantiation of (14), therefore, must be within the intended range of abstraction. Thus,

"CS135" . ()x:Person)(x is a student)

is considered as a nonsignificant symbol because

CS135 is a student

is not within the range of generalization intended by (14).

Note that the above concept of type is akin to the one used in logic, particularly in the theory of types [29], where logically paradoxical symbols are discarded as non-significant due to type violation. However, it is different from common notions of type in programming language and database literature (e.g., in [16]), where a type is defined as a category of objects. Russell [30] characterizes his theory of types as a theory of symbols. In programming language the theory of type is a theory of objects. A more detailed discussion about the concept of type in ADS is presented elsewhere [19].

3.4 Transparent Quote

ADS has two kinds of quotation marks: single (') and double (") quotation marks. (Note that in this paper the single quotation marks are also used in our English exposition.) When a symbol with free occurrences of variables is quoted by single quotation marks, the occurrences remain free. When such a symbol is quoted by double quotation marks, the occurrences are bound. The single quotation mark pair is called a transparent quote.

Without the transparent quote the expressive power of abstraction is somewhat limited. Consider a generalization of the following two data:

(20) "John" is the name of John

(21) "Dave" is the name of Dave.

If we try to generalize the above symbols into

(22) ()x:Person)("x" is the name of x),

then the first occurrence of 'x' in the abstraction body is not free, and the intensional instantiation of (22) based on 'John' will yield '"x" is the name of John', instead of (20). With the transparent quote, the proper generalization of (20) and (21) is

()x:Person)('x' is the name of x)

whose intensional instantiations will yield (20) and (21).

3.5 Description Operator

In logic, '(ιx)P(x)' means 'the x such that P(x) is true' where P(x) is a predicate. 'ι' is called the definite description (iota) operator. In ADS, for an arbitrary predicate P(x), we define the iota operator by

(ιx)P(x) ≡ "Truth" ⁎ ()/x)P(x).

Therefore, by the definition of '⁎', for an arbitrary symbol a,

(ιx)P(x) ≡ a iff P(a) :=: "Truth".

We call the iota operator the description operator. Thus, '(ιx)P(x)' means 'an x such that P(x)', and it is nonsignificant only when P(x) is true of nothing. The description operator is an extended form of the definite description operator in symbolic logic and may well be called the indefinite description operator.

3.6 Levels of Abstraction

By the mechanisms of the lambda operator and the inverse lambda operator, any symbol can be generalized or discriminated into an abstraction, including any abstraction descriptor. An abstraction on abstraction descriptors is called a second order abstraction, and a descriptor denoting a second order abstraction is called a second order descriptor. Higher order abstractions and higher order descriptors can be similarily defined.

Consider, for example, the following set of statements:

(23) John takes CS135

(24) ()x:Course)(John takes x)

(25) ()y:Student)()x:Course)(y takes x)

(26) ()z:Relation)()y:Student)()x:Course)(y z x).

Symbol (24) is a generalization of (23), (25) is a generalization of (24), and (26) is a generalization of (25). Abstraction (24) may be used to represent the set of courses being taken by the student John. Symbol (25) may represent the courses-taken attribute of a student. It may be used to represent the concept of enrollment in terms of the set of courses taken by each student. Symbol (26) expresses the most abstract concept derivable from statement (23). Abstractions (24) through (26) demonstrate that a nested application of the lambda operator facilitates arbitrary levels of abstraction on symbols with a simple parsimonious construct.

4. Database Concepts

In the previous sections we discussed some of the important concepts about symbols. In this section and the next, we will discuss the relationship between symbols and a database.

4.1 Judgment

A database is a depository of symbolic representations of user judgments. A judgment is an act of representing some aspect of reality in symbolic form, usually as a declarative sentence which asserts the proposition expressed by the sentence. A judgment is a building block of a user's knowledge and view. A judgment has been made only when a symbol corresponding to it has been produced and asserted.

For example, 'John is a student' may be the result of a judgment made by user A, about a person John, based on A's concept of student. If user B does not agree with the judgment, B probably has a different concept of John and/or student than A does, i.e., B's interpretation of the symbols 'John' and 'student' is different from that of A. User B may object to entering the statement into the database because it may result in an inconsistent database (or inconsistent view of reality) from B's point of view.

4.2 Unstratified Database

For efficient and precise communication among users, it is essential that the interpretation rules (semantics) of the symbolism used for representing judgments in the database be shared and clearly agreed upon. One mechanism for such sharing is the database itself. A database may carry symbolic representations of judgments about the symbolism, as well as about reality. Thus, the database itself becomes a part of reality, and a database manager

who makes judgments about the database symbolism (e.g., changes in the syntax and semantics of the type), becomes a user. When a database employs the same symbolism to represent judgments about the symbolism as about reality, we call it an unstratified database.

The part of an unstratified database that represents the reality outside of the database is called the object database and the part that represents judgments about the object database is called the meta database. The concept of a meta meta database and so on can be similarly defined.

For example, consider the following symbols representing different kinds of user judgments:

(27) John is a student

(28) "John" is a first name

(29) "Is a first name" is a predicate symbol.

Statement (27) belongs to an object database, (28) to a meta database and (29) to a meta meta database.

4.3 Intensional and Extensional Judgments

The stratification of a database into an object database, a meta database, and so on, classifies user judgments by the levels of languages used to represent them. Another classification of user judgments which is independent of the above is that of intensional and extensional judgments. A judgment is intensional or extensional depending on whether the subject of judgment is represented by the sense of a symbol or by the denotation of a symbol. An intensional judgment contributes to general knowledge and an extensional judgment contributes to specific knowledge [25]. The first is conceptual, and the second is factual and less abstract than the first. Intensional judgments

are usually less volatile than extensional judgments.

For example, statements (27) through (29) represent extensional judgments referring to a specific person or specific symbols. The following are examples of intensional judgments, corresponding to (27) through (29):

(30) A student is a person

(31) A name is a sequence of less than ten letters

(32) An expression following the subject phrase in a sentence is a predicate.

They refer to general concepts of person, student, and so forth. These judgments can be made independently of extensional knowledge. For example, statement (30) represents a judgment about properties of a student, assuming such an entity exists. Even if no student is currently known to exist, the judgment is meaningful because it establishes a relationship between the concepts of student and person.

4.4 Semantic Consistency

The fundamental law of semantic consistency states that the extensional judgments must be consistent with the intensional judgments. For example, statement (28) is consistent with (31) because 'John' has less than ten letters. Similarly, (29) is consistent with (32). Statement (27) is consistent with (32), assuming that the database contains the following extensional judgment:

(33) John is a person.

Even without (33) the database can be considered as consistent in the sense that no logical contradiction is derivable from the database. Since there is no qualification about the concept of person in the above database,

anything can be judged to be a person without violating the consistency rule. Note that we assume that the absence of a statement in the database does not necessarily imply that its negation is valid in reality, i.e., we adopt the open world assumption [14].

4.5 Denotational Database

A fundamental limitation of ADS is that it supports only denotational databases in the sense explicated below.

A database is denotational if any symbol A in the database, that is not quoted, can be substituted by another symbol B without violating the semantic consistency, as long as A and B are equivalent (i.e., have the same denotation). For example, consider a situation in which the following statements are true, i.e., all of them have the denotation, Truth:

(34) The instructor of CS360 is Cox

(35) The advisor of Smith is Cox

(36) It is necessary to include in the meeting
the advisor of Smith

(37) It is not necessary to include in the meeting
the instructor of CS360.

Since 'the instructor of CS360' and 'the advisor of Smith' have the same denotation by the propositions expressed by (34) and (35), if the database containing (34) through (37) were denotational, then substituting 'the instructor of CS360' for 'the advisor of Smith' in (36) would not make the database inconsistent. However, the substitution makes (36) a contradiction of (37). Therefore, the database cannot be denotational.

In general, judgments involving the notions of needing, wanting, knowing, believing and so on cannot be represented in a denotational database. They require a logistic system by which deductions on the sense (concept) become possible. If the semantic consistency of judgments is to be maintained by a computer, a database must be denotational, because there is no known established formal theory of concept at the present time (see [23], [27] and [36] for some efforts towards this goal).

ADS is a data model for denotational databases.

4.6 Fundamental Concepts Representing User Judgments

A judgment may have different kinds of symbolic representations. The choice of symbolism (data language) is not free from imposing a particular bias on the modeling of reality.

It is important to minimize such a bias for a universal data language. Consider the user judgment represented in English by 'John is a student'. Assuming the standard mathematical notation, the same judgment can be represented in at least two different ways:

(38) John ϵ Student

(39) Student(John) = Truth

where Student is defined as a set for (38) and as a function for (39). Expression (38) implies that the concept of student is modeled as a set of objects. Any attempt to interpret the expression (i.e., specifying the semantics of the data language) requires the notion of set. On the other hand, the interpretation of (39) requires modeling the concept of student as a function from objects to logical values. Its semantic specification requires the notion of function as a primitive concept. While sets and functions are

mathematically equivalent (i.e., the extension of a function is definable by a set of ordered pairs, and a set is definable by its characteristic function), the concept of function is not the same as the concept of set. Thus, the two expressions utilize two different concepts, set and function, in representing the same user judgment.

ADS uses abstraction on symbols for representing both sets and functions. An ADS representation of 'John is a student' is:

John . Student.

The concept of abstraction on symbols in ADS provides a unifying and parsimonious mechanism for the concepts of set and function.

5. Database Use of Symbols

In ADS what can be described is what can be named.

5.1 Naming and Describing

When a database user represents a judgment in symbolic form, he may refer to an object in two different ways, by naming the object or by describing the object.

Naming by a proper name (e.g., 'John'), allows us to refer to an object without necessarily involving its structure, properties, or relationships to other objects. A name hides every aspect of the object except for its identity. On the other hand, describing by a description (e.g., 'the best student in CS135') provides a method of modeling (or characterizing) the object either through its structure, its properties, or through its relationship to other objects. A description hides selective aspects of the object. The selection reflects a view of the author of the description. Both naming and describing are processes of abstracting reality into symbolic

representations. For other views of naming, see Church and Carnap ([7] and [2], respectively).

In naming, any symbol can be a name, i.e., the association between a name and its meaning is arbitrary [3]. There is no logical necessity to call some person by a particular name, such as 'John'; it can be any other symbol, such as 'ABC'. The structure of a name has no bearing on its meaning. It follows that (i) the meaning of a name must be defined externally; there is no intrinsic meaning associated with a symbol used as a name, and (ii) a name, as a proper name, has a denotation (what it names) but no sense.

In describing, the choice of symbols and their arrangement (syntax) cannot be arbitrary. For example, 'the best CS135 in student' is nonsensical. The syntactic structure of a description not only determines the object it describes (its denotation) but also determines the way of identifying the object (its sense). A description is called a descriptor.

5.2 Attributive and Referential Use of Descriptors

In general there are two different kinds of descriptor usage, namely an attributive use and a referential use [21]. Consider the descriptor 'the best student in CS135' in the following statement:

(40) The best student in CS135 is smart.

The statement is ambiguous because it can be interpreted in two different ways:

(41) Whoever is the best student in CS135 is smart

(42) John who happens to be the best student in CS135 is smart.

If (40) is intended to mean (41), then the descriptor is used attributively where the sense of the descriptor is used for obtaining the meaning. If (40)

is intended to mean (42), then the descriptor is used referentially, where the denotation of the descriptor is used.

In ADS a descriptor is always used referentially, i.e., whenever the descriptor is evaluated, its denotation is taken as its meaning. Thus, (40) means (42) in ADS. This is consistent with the assertion made in Section 4.5 that the ADS data model supports denotational databases. However, this does not imply that ADS descriptors are devoid of sense. We assume that every descriptor has a unique sense.

5.3 Definition of Names

A definition of a name is an association between a name and a descriptor. There are two possible ways of establishing such an association: the symbol can be defined as naming the sense of the descriptor or as naming the denotation of the descriptor. In ADS these two types of name definitions are distinguished as follows:

(43) ABC == the best student in CS135

(44) DEF := the best student in CS135.

Definition (43) defines 'ABC' as a name expressing the same sense that the descriptor expresses, and (44) defines 'DEF' as a name denoting the same denotation that the descriptor denotes in the current context. The symbol 'ABC' becomes a synonym for 'the best student in CS135' (i.e., having the same intension). The symbol 'DEF' becomes an equivalent symbol to 'the best student in CS135' and becomes a proper name for the best student in CS135 at the time when (44) is performed. Thus, the definitions have the same effect as letting the following statements be valid:

"ABC" ≡ "the best student in CS135",

and "DEF" :=: "the best student in CS135".

Definition (43) is an intensional definition of the name 'ABC' where the sense (intension) of the descriptor defines the intension of the name. Statement (44) is an extensional definition of name 'DEF' where the denotation (extension) of the descriptor defines the extension of the name but does not define the intension. The descriptor in an intensional definition is called the intensional descriptor of the name, and it can be referenced as a symbol in the ADS data language by prefixing the name with '@'.

5.4 Intensional and Extensional Use of Names

Since all descriptors are used referentially in ADS, intensionally defined names are also used referentially. They refer to the denotation of the defining intensional descriptor evaluated at the time of name usage. In contrast, extensionally defined names refer to the denotation of the defining extensional descriptor evaluated at the time of name definition.

When a name is defined both intensionally and extensionally, there may be possible ambiguity over what the name refers to. For example, assume that the following intensional and extensional definitions of the name 'ABC' are asserted when John is the best student in CS135.

(45) ABC == the best student in CS135

(46) ABC := John.

Then, the statement

(47) ABC takes EE280

is ambiguous as to whether it is synonymous with (48) or only equivalent to (49):

(48) The best student in CS135 takes EE280

(49) John takes EE280.

When (47) is intended to mean (48), we say the name 'ABC' is intensionally

used. When it is intended to mean (49), the name is extensionally used. Note that regardless of whether a name is used intensionally or extensionally, a name in ADS is used for referencing an object, not a concept, i.e., for referencing the denotation of some descriptor.

In order to eliminate such ambiguity, ADS adopts the following rule:

> Any extensional usage of a name is prefixed by the symbol '#', and the name by itself indicates an intensional usage.

Thus, the following is implied by (45) and (46), respectively:

"ABC takes EE280" ≡ "The best student in CS135 takes EE280", and

"#ABC takes EE280" :=: "John takes EE280".

5.5 Consistency in Name Definitions

When a name is defined both intensionally and extensionally the definitions cannot be independent. If a name represents some entity in the user's perception of reality, it is natural to expect that the object denoted by a name be consistent with the concept expressed by the name. This is the ADS version of the fundamental law of semantic consistency.

When this fundamental law is enforced, it is compatible with the following observation about how a name is used in ADS: The intension of a name specifies what the name can possibly represent and the extension of the name specifies what the name does actually represent.

In order to uniformly preserve the fundamental law of consistency, ADS requires that every name be defined intensionally first before any extensional definition. In other words, what objects The name can denote must be defined before the name is actually used to denote a specific object. This is analogous to the requirement in a programming language that a variable type

must be declared before a data value can be assigned to the variable.

All names in ADS have an intension while some have both an intension and an extension. No name has only an extension. When a name has no extension, a name with the prefix '#' is nonsignificant but is not nonsensical.

5.6 Statement (Boolean Descriptor)

A descriptor is called a statement if it denotes a logical value (Truth or Falsehood) and expresses a proposition as its sense. Thus, in ADS, a statement is called a boolean descriptor. For example, 'John is a student' expresses the proposition that John is a student and denotes Truth if John actually is a student.

One of the most primitive and fundamental forms of a statement is a copula connecting a subject A and a complement B as in

A is B

where A is a descriptor of an object and B is a descriptor of an object or a property. When B describes an object, the descriptor B and the copula are used equatively (or the statement is equative), otherwise they are used predicatively (or the statement is predicative). For example, consider the following statements:

(50) The best student in CS135 is John

(51) John is a student

(52) John is the best student in CS135.

Statement (50) is equative, expressing the fact that the person referred to by the symbol 'the best student in CS135' is identical to the person referred to by the symbol 'John'. Statement (51) is predicative, expressing the fact that the person John has all attributes necessary for being a student such as being

properly registered, having paid a tuition, and so on. Statement (52) is ambiguous as to whether it is equative or predicative.

In ADS, an equative usage of the copula is represented by the symbol '=' as in the following representation of (50):

The best student in CS135 = John.

The equality symbol represents the identity relation on objects. It belongs to the object language (the ADS language) rather than the meta language (English).

In ADS, there are two different ways of representing a predicative usage of the copula such as in (51): by the generalization operator or by the description operator. Assume that the concept of student can be decomposed into more primitive concepts such as a person who has paid tuition, i.e., assume that (51) is logically equivalent to:

John has paid tuition.

Then, we may define a generalization named Student as:

Student == (λx:Person)(x has paid tuition)

and represent (51) by its (intensional) instantiation as:

(53) "John" . Student.

In this way the copula 'is' is represented by the instantiation operator '.'. If the abstraction Student is interpreted as a certain class of person, then (53) expresses set membership. If it is interpreted as a predicate function, then (53) expresses a function application.

The other way of representing a predicative usage of a copula is to consider it as the special case of an equative usage, equating the subject with some object that has all the properties specified by the predicate. For example, (51) expresses the identification of John with some person who has

all the necessary properties for being a student. Assuming again that any person who paid tuition is a student, we may define an indefinite object student as:

student == (ι x:Person)(x has paid tuition).

Then we can represent (51) by:

John = student.

In ADS, a statement (boolean descriptor) is either equative with '=', or predicative with '.' (or with '$_*$'), or a logical composition of these statements with the standard logical operators, '∧' (and), '∨' (or), '~' (not), '∀' (for all), and '∃' (for some).

5.7 Assertion of Statement

Database users interact with a database either by asserting a statement or by de-asserting a statement which had been previously asserted (i.e., cancelling a previous assertion). The logical assertion symbol '⊢' indicates an assertion, and the symbol '⊣' indicates a de-assertion as in:

(54) ⊢ John is a student

(55) ⊣ John is a student.

Note that a de-assertion is different from a negative assertion. De-assertions decrease the amount of information obtainable from a database, while negative assertions increase it. After the judgment represented by (55) is made, it is not known whether John is a student or not. After entering the following negative assertion,

(56) ⊢ John is not a student

it is known that John is not a student. Assertion (56) is inconsistent with (54), but is consistent with (55).

In ADS, all assertions are either intensional or extensional definitions of names with the form:

⊢ <name> == <descriptor>

or ⊢ <name> := <descriptor>.

Here '<name>' represents an arbitrary name and '<descriptor>' represents an arbitrary descriptor. Therefore, in order to enter assertion (54), a user would decompose it into two parts; first, name the proposition then define the extension of the name to be true:

(57) ⊢ FACT == John is a student

(58) ⊢ FACT := Truth.

5.8 Constraints and Transactions

When a boolean name is extensionally defined (e.g., (58)) the database system first checks the semantic consistency with the intensional definition (e.g., (57)), i.e., it checks whether the statement 'John is a student' will be evaluated to be true under the current database state. Once the extensional definition is accepted it becomes a fact to the database system and participates in future semantic consistency checks. Any assertions made to the database system must be consistent with the set of facts known to the database system. Thus, the set of extensionally defined boolean names in ADS are called constraints. Note that a constraint can be activated or deactivated by asserting or de-asserting its extensional definition. Also note that a constraint can express a general fact such as:

Course-Limit == No student takes more than 6 courses

Course-Limit := Truth.

A constraint may be local, involving a single name, or global, involving more than one name. When a name in a global constraint acquires a new definition, the database may become inconsistent. It follows that a single assertion or de-assertion (updating a name definition) may not preserve semantic consistency.

A transaction is a sequence of assertions and/or de-assertions that preserves semantic consistency, and is identified by '<' and '>' surrounding the sequence. The database system temporarily suspends consistency checking until the end of the transaction. If the transaction results in an inconsistent state, the database state prior to processing the transaction is retained. For example, consider a global constraint BALANCE defined as follows:

```
⊢       ADD  == the number of courses added
⊢       ADD  := 25
⊢       DROP == the number of courses dropped
⊢       DROP := 25
⊢       BALANCE == (#ADD = #DROP)
⊢       BALANCE := Truth.
```

Now, in order to increase the number of added and dropped courses without violating the constraint BALANCE, we have to use a transaction as follows:

```
<
⊢       ADD  := #ADD + 1
⊢       DROP := #DROP +1
>.
```

In summary, we list the symbols used in Sections 2 through 5 along with the fundamental concepts they represent:

≡	synonymy (2.2)
:=:	equivalence (2.2)
/ /	symbol significance (2.4)
" "	quote (2.5)
[ ]	unquote (2.6)
→ ;	conditional control (2.8)
λ	generalization operator (3.1)
γ	discrimination operator (3.1)
.	intensional instantiation (3.2.1)
*	extensional instantiation (3.2.2)
:	variable type specification (3.3)
' '	transparent quote (3.4)
ι	description operator (3.5)
==	intensional definition of name (5.3)
:=	extensional definition of name (5.3)
@	intensional descriptor of name (5.3)
#	extensional use of name (5.4)
=	equative statement (5.6)
∧	and (5.6)
∨	or (5.6)
~	negation (5.6)
∀	for all (universal quantifier) (5.6)
∃	for some (existential quantifier) (5.6)
⊢	assertion of statement (5.7)
⊣	de-assertion of statement (5.7)
< >	transaction (5.8).

In Sections 6 through 8, we will illustrate how these concepts are incorporated into the data language and how they can be used to represent user perceptions of reality.

6. An Overview of ADS

In this section we summarize the basic components of the ADS language. A preliminary version of the ADS language appeared in [9], and a formal specification of ADS is given in [18].

A database system is a communication mechanism for a community of users to share their views of reality. The users interact with the database system through a symbolism whose syntax and semantics are well defined and known to every user in the community. The depository of the symbols that represent collectively the history of user judgments is a database. The content of the depository is a database state. The rest of the database system is a database manager, responsible for enforcing database integrity by rejecting any judgments which are inconsistent with the current database state.

6.1 The ADS Database System

The ADS data language can be considered as an input specification language for the ADS database system. A single sequence of commands is input, to the ADS database system, and a single sequence of responses is output from the system. Commands may be either update or query commands.

The ADS database system utilizes an interpretation function (of the data language) with the command and the current database state as input and the response and the next database state as output. The command is evaluated

within the context defined by the current database state. The command may define a new context by changing the database state. If the command is a query command, the appropriate information is extracted from the current database state and becomes the response; the state is left unchanged. If the command is an update command, the potential next state is created and checked for consistency. If it is a consistent state, it becomes the next state and the command is accepted; otherwise, the state is left unchanged and the command is rejected.

When a sequence of update commands is entered as a transaction, the database state immediately preceding the transaction is saved, in preparation for the contingency that the processing of the transaction may result in an inconsistent database state. Such a transaction will be rejected by the database system and the saved state will be unsaved (restored) as the next state; consequently no change occurs to the database state. If the resulting state is a consistent state, then it becomes the next database state immediately after the transaction is accepted. During the processing of the transaction, consistency checking is suspended.

6.2 Objects

Query and update commands are expressed in the ADS language through the use of names and descriptors of ADS objects. An object is what a user can name and describe in ADS. The users represent their knowledge about reality in terms of ADS objects and the relationships among them.

There are three categories of ADS objects:

1) Symbols. They are used to represent both entities in reality and expressions in the data language. The syntactic categories of the

ADS language, such as variables, names, descriptors, and commands, are all subcategories of the symbol objects. The operations on the symbol objects are composition (concatenation) and decomposition (deconcatenation).

2) Logical Values. They are the result of evaluation of logical expressions, i.e., Truth and Falsehood. The standard logical operations are assumed.

3) Abstractions. They are used to represent sets of entities, functions (attributes) on entities, predicates (properties) on entities, sets of sets of entities, and so on. The operations on abstraction objects are instantiation and abstraction. Instantiation applied to an n-th order abstraction yields an (n-1)st order abstraction, where $n>0$. The abstraction operation applied to an n-th order abstraction yields an (n+1)st order abstraction, where $n\geq 0$. A 0-th order abstraction is defined to be a symbol.

Among these categories, only the symbol category is discussed in detail below.

6.3 Symbols

Any pattern can be a symbol provided that there exists an interpretation rule associated with it. Nothing can be a symbol without such an interpretation rule. Logically speaking, any infinite decidable set of patterns can be the set of symbols. We have chosen the set of binary trees as the set of symbols for the data language. We define and denote the set of binary trees as follows:

(i) 0 is a binary tree (the null tree).

(ii) If a and b are binary trees, then so is 1ab.

(iii) Nothing else is a binary tree.

For example, the following is a set of binary trees:

{0, 100, 10100, 11000, 1100100, 1010100, 1011000, ... }

The set of ADS symbols (binary trees) is decidable (because it is context-free), and patterns are uniquely deconcatenable, i.e., when two patterns (binary trees) of arbitrary length are concatenated, it is always possible to separate them uniquely. (This property plays an important role in designing VLSI storage devices for the ADS symbols.)

The ADS symbols are isomorphic to the restricted LISP S-expressions with NIL as the only atomic symbol [22]. Because parsing and recognizing binary trees is difficult for humans, we use English-like symbols as a meta-language (corresponding to M-expressions in LISP) and assume that an appropriate translation rule exists from the meta-language to the ADS symbols. The ADS symbols consist of the null symbol (the null tree) denoted by '<>' in the meta-language, and the compound symbols denoted by '<a,b>' where a and b are arbitrary ADS symbols. The identity of two symbols is represented by '=' in ADS. As in LISP, a list of symbols is defined, for example, as:

(a,b,c,d) ≡ <a,<b,<c,<,d,<>>>>>.

All variables in the ADS language range over the ADS symbols. The variables are used in conjunction with abstraction operators (generalization, discrimination, and indefinite description) and quantification operators (universal and existential). Thus, abstractions in ADS are always on symbols.

6.4 Database Objects

The ADS database state consists of a set of symbols (binary trees) of the following form, each of which is called a database object,

(name, intensional descriptor, extensional descriptor),

where each name in the database is unique. The intensional descriptor defines the sense of the name, and the extensional descriptor defines the denotation of the extensionally used name.

Note that the logical structure of the database state is similar to that of the symbol table in a programming language interpreter whose entries have, in general, the form,

(identifier, type, value),

where the type specifies what value can be assigned to the identifier and the value specifies what is currently assigned to the identifier. Also note that a database object in ADS corresponds to the first three components of an elementary datum as defined by Langefors [20]:

(object name, object properties, property value, time).

A database state is consistent if for each database object in the database state the extensional descriptor is consistent with the intensional descriptor based on the fundamental law of semantic consistency; otherwise, it is inconsistent.

Entering the command

⊢ name == descriptor

creates the new database object

(name, descriptor, undefined)

provided that there is no database object in the database which has the same name.

Entering the command

⊢ name := descriptor

modifies a database object from the form

(name, descriptor1, descriptor2)

to the form

(name, descriptor1, descriptor).

The command will be rejected if there is no database object in the database which has the same name, or if the command will lead to an inconsistent database state.

Similarly, entering a command de-asserting the intensional definition of a name will delete the entire database object. A command de-asserting the extensional definition of a name changes the third component of the named database object to 'undefined'.

7. School Database

One of the most important and unique features of the ADS data model is its capability of integrating the intensional database with the extensional database in a flexible manner. The capability is important because the integration of knowledge about the possible world (represented by intensional data) and the actual world (extensional data) is very common in a user's perception of reality. The flexibility in representing interactions between the two enhances the expressive power of the ADS data language. In this section, we illustrate this capability and the fundamental law of semantic consistency in ADS. We also illustrate how the abstraction capabilities of the ADS language can be used for representing user judgments and queries. In order to cover more variety of ADS language constructs, we view the world in

terms of entity types and attribute functions as in [31], rather than a set of relations. An ADS representation of the relational view of the world will be given in the next section.

Numbered user judgments and queries appear, one by one, first in the ADS language, and then in English. We will consistently omit the quotation marks on both sides of '≡' and ':=:' for brevity.

Assume that all proper names, such as 'Cox', 'CS135', and so on, are already defined intensionally, each name denoting some symbol which is an internal representation (surrogate in [17]) of a person or a course in reality. For example, Cox == "Cox", CS135 == "CS135", etc. Similarly, 'T' denotes the symbol 'Truth', and 'F' denotes 'Falsehood'.

7.1 Entity Types

Let us suppose that, as a user, we identify the need for introducing an entity type called Person, and that nothing is known about the entity type with respect to its membership and associated attributes. At this point we enter the judgment into the database expressed as:

(59) ⊢ Person == ()x)T — Accept

Any object can be a person.

The first-order abstraction Person represents the set of all possible persons. The database acknowledges the assertion by the response 'Accept' given at the right.

In ADS the question mark followed by a descriptor represents a query, requesting the object denoted by the descriptor.

(60) ? Cox.Person — Yes

Can Cox be a person?

The response is again given at the right expressed in a meta-language (in this paper we are using English). Since

$$\text{Cox.Person} \equiv \text{Cox.}(\lambda x)T :=: T,$$

the response from the database is a display of the logical value, Truth, in the meta-language. We choose to display Truth by 'Yes' and Falsehood by 'No' to make the interaction more natural in English.

At this point, 'Person' is not extensionally defined, i.e., '#Person' is nonsignificant, which corresponds to the fact that no person is actually known yet. Therefore,

(61) ? Cox $_{*}$ #Person — Reject

Is Cox a person?

Now, suppose that the person Cox becomes a known person, and that the fact is to be entered into the database. The following extensional definition will suffice:

(62) $\vdash$ Person := $(\lambda x)(x=\text{Cox} \rightarrow T)$ — Accept

Cox is a person.

(63) ? Cox $_{*}$ #Person — Yes

Is Cox a person?

Definition (62) associates the name 'Person' with the denotation of the descriptor '$(\lambda x)(x=\text{Cox} \rightarrow T)$'. The response component of (63) is 'Yes' because,

$$\text{Cox} \,{}_{*}\, \#\text{Person} :=: \text{Cox} \,{}_{*}\, (\lambda x)(x=\text{Cox} \rightarrow T) :=: (\text{Cox}=\text{Cox} \rightarrow T) :=: T.$$

Extensional definition (62) is accepted by the database, because it is consistent with the intensional definition (59) in the following sense:

For an arbitrary symbol a,

a $_{*}$ #Person :=: T implies a . Person :=: T.

In other words, if the object denoted by a, is a person, then the object must be one of those that can be a person (note the distinction between <u>is</u> and <u>can be</u>). This is the definition of the fundamental law of semantic consistency applied to the abstraction name 'Person'.

It should be noted that the law, as stated above, is independent of what the abstraction represents, e.g., whether Person represents a set or a propositional function. If Person represents a set, then the law can be translated to:

The set represented by #Person must be a subset of the set represented by Person.

Similarly if Person represents a propositional function, then the law can be translated to:

The function represented by #Person must be a restriction (a partial function) of the function represented by Person.

7.2 <u>Update of Entity Types</u>

If another person, Gillett, becomes known, the extension of the name 'Person' must be updated:

(64) $\vdash$ Person := ()x)(x=Gillett $\rightarrow$ T; x $_{*}$ #Person) Accept

Gillett is also a person.

(65) ? Cox $_{*}$ #Person Yes

Is Cox a person?

Note that in (64) the new extension of 'Person' is defined in terms of the old extension of the name. Since prior to (64),

#Person :=: (λx)(x=Cox → T),

the response of (65) is justified by:

Cox ∗ #Person :=: Cox ∗ (λx)(x=Gillett → T; x ∗ (λx)(x=Cox → T))

:=: (Cox=Gillett → T; Cox ∗ (λx)(x=Cox → T))

:=: Cox ∗ (λx)(x=Cox → T)

:=: T.

ADS has an abbreviation for incrementally updating extensional definitions like (64) as follows:

⊢ Gillett ∗ #Person := T Accept

In general, for arbitrary symbols a and b, and for an arbitrary first-order generalization name A,

⊢ a ∗ #A := b ≡ ⊢ A := (λx)(x=a → b; x ∗ #A).

Using this syntax for updating, we can enter four more persons into the database:

(66) ⊢ Kimura ∗ #Person := T Accept

⊢ John ∗ #Person := T Accept

⊢ Dave ∗ #Person := T Accept

⊢ Kathy ∗ #Person := T Accept.

Note that the above four updating assertions can be replaced by the following one assertion if those four are known in a group:

⊢ Person := (λx)(x=Kimura∨x=John∨x=Dave∨x=Kathy → T;

x ∗ #Person) Accept

The entity type Course, Student, and Faculty can be defined in a similar way as for Person:

(67) ⊢ Course == (λx)T Accept

Any object can be a course.

(68) ⊢ Course := (λx)(x=CS135 ∨ x=CS236 ∨ x=CS301 ∨ x=CS360 → T)

Accept

CS135, CS236, CS301, CS360 are known (existing) courses.

(69) ⊢ Student == (λx:#Person)T Accept

Any person (any object that is known to be a person) can be a student.

(70) ⊢ Student := (λx:#Person)(x=John ∨ x=Dave ∨ x=Kathy)

Accept

John, Dave, and Kathy are students.

(71) ⊢ Faculty == (λx:#Person)T Accept

Any person can be a faculty member.

(72) ⊢ Faculty := (λx:#Person)(x=Cox ∨ x=Gillett ∨ x=Kimura)

Accept

Cox, Gillett and Kimura are faculty members.

Note the difference between (69) and the following:

(73) ⊢ Student == (λx:Person)T Accept

Any object that can be a person can be a student.

The difference can be illustrated by considering the query:

(74) ? Steve . Student Yes with (73), Reject with (69)

Can Steve be a student?

The responses are justified because

Steve . Person :=: T,

and Steve ∗ #Person :=: F.

In other words, 'Steve' is a symbol of type Person but it is not of type #Person, and hence (74) produces a type violation if (69) is in effect.

Accordingly, with (73) the following extensional definition is acceptable, while it is not with (69):

⊢ Steve ∗ #Student := T Accept

Steve is a student.

Note also that

(75) ? CS135.Student Reject (Unanswerable)

Is CS135 a student?

because 'CS135' is not a symbol of type Person, and 'CS135.Student' is nonsignificant due to a type violation. The database rejects the query as unanswerable. For testing a potential violation of type, the following query will suffice:

(76) ? /"CS135.Student"/ No

Is 'CS135.Student' significant?

7.3 Consistency Checking

We show here that a local update may involve a global consistency check. Definitions (59) and (67) allow the two entity types Person and Course to overlap completely. If we know that courses are different from persons, then that knowledge can be represented in the following definition of Course in place of (67):

(77) ⊢ Course == ()x)(x ∗ #Person → F; T) Accept

Any object that is not a (known) person can be a course.

Even though definition (77) explicitly restricts only 'Course', it also implicitly restricts 'Person' as well. Consider the following update assertion attempted after (59), (77) and (68) have been entered:

(78) $\vdash$ CS135 $_{*}$ #Person := T Reject

CS135 is also a person.

The response is 'Reject' because the resulting database state would violate the fundamental law of semantic consistency with respect to the name 'Course', rather than the name 'Person':

CS135 $_{*}$ #Course :=: T but CS135 . Course :=: F.

If it is desirable to make a similar explicit restriction on Person for symmetry, the intension of 'Person' can be modified by the following transaction (ref. 5.9):

(79) <

$\dashv$ Person == ()x)T

$\vdash$ Person == ()x)(x $_{*}$ #Course $\rightarrow$ F; T)

> Accept

7.4 Constraints

Another way of representing the mutual exclusion of #Person and #Course is to enter the following pair of assertions, which constitute a constraint (ref. 5.9) on the extensions of 'Person' and 'Course':

(80) $\vdash$ Course-Constr == ($\forall$x)(x $_{*}$ #Person $\rightarrow$ ~(x $_{*}$ #Course); T)

Accept

For any object, if it is a person, then it is not a course.

(81) $\vdash$ Course-Constr := T Accept

Let Course-Constr be true.

Assertion (81) is accepted by the database because after it is entered the database state is consistent with (80) in the following sense:

#Course-Constr :=: T implies Course-Constr :=: T.

This is the definition of the fundamental law of semantic consistency applied to the name 'Course-Constr'. Once (81) is accepted, any update on the extension of either Person or Course must preserve the denotation of 'Course-Constr' to be Truth; otherwise the law would not hold. Thus, in general, an extensional definition of a name for a logical value imposes a constraint on the database state.

7.5 Open Queries

All queries introduced so far have been closed queries, those which can be answered by either 'Yes' or 'No'. Open queries are represented in ADS by the question mark followed by an indefinite descriptor or a first-order abstraction descriptor (ref. 3.1). For example,

(82) ? $(\eta x{:}\#Person)(x_{*}\#Student)$ John (Dave, Kathy)

Which person is a student?

asks for any one person who is a student. The response could be John, Dave or Kathy, but only one of them should be given. The query is synonymous to:

? $T_{*}\ (/x{:}\#Person)(x_{*}\#Student)$

Who is a person x such that 'x is a student' is true?

If the discrimination descriptor by itself follows the question mark, without the extensional instantiation 'T_{*}', then, by ADS convention, the query asks for all possible significant results of such extensional instantiations, thus:

(83) ? $(/x{:}\#Person)(x_{*}\#Student)$ John,Dave,Kathy

Which persons are students (who are the students)?

This illustrates how the totality of set membership can be determined in ADS.

7.6 Attribute Functions

When we make a judgment that a student may have any faculty member but himself as his advisor, an attribute function, advisor, can be defined as follows:

(84) ⊢ advisor == ()x:#Student)(η y:#Faculty)(~ x=y) Accept

Any faculty member can be the advisor of any student but himself.

Note that from (69) and (71) any person can be both a student and a faculty member concurrently.

In order to ask who can be Dave's advisor, the following query will suffice:

(85) ? Dave.advisor Cox (Gillett, Kimura)

Who can be Dave's advisor?

The above query is synonymous to

? (η y:#Faculty)(~ y=Dave)

requesting one of the faculty members who is not Dave.

Later, when a particular advisor-advisee assignment is made, the extension of 'advisor' can be entered as follows:

(86) ⊢ advisor := ()x)(x=John → Gillett; x=Dave → Kimura) Accept

John's advisor is Gillett and Dave's advisor is Kimura. The above extensional definition is consistent with (84) in the following sense:

For arbitrary symbols a and b,

a_{*} #advisor :=: b implies a . advisor :=: b.

This is the fundamental law of semantic consistency applied to the attribute function name 'advisor'.

7.7 Individual Entities

Suppose that it is required to assign Kathy an advisor who is chosen from the faculty members who currently have no advisee. First we establish the concept of an available faculty member for advising, and then we choose one specific faculty member as the available advisor. We can then assign the faculty member to Kathy:

(87) ⊢ Available-Advisor == (↿x:#Faculty)(∀y:#Student)
(/'y ⁎ #advisor'/ → ~(x = (y ⁎ #advisor)); T) Accept

A faculty member who has currently no advisee
can be an available-advisor.

(88) ⊢ Available-Advisor := Cox Accept

Cox is an available advisor.

(89) <

⊢ Kathy ⁎ #advisor := #Available-Advisor

Kathy's advisor is the current available advisor.

⊣ Available-Advisor:= Available-Advisor

The current available advisor will no longer be an available advisor.

> Accept.

The extensional definition of assertion (88) is consistent with assertion (87) because the following condition is satisfied:

For arbitrary symbol a,
(a = #Available-Advisor) :=: T implies
(a = Available-Advisor) :=: T.

This is the fundamental law of semantic consistency applied to the symbol name 'Available-Advisor'.

Note that according to definition (87), when Cox is Kathy's advisor, Cox cannot be an available advisor. Therefore, it is necessary to de-assert (88) and assign Cox as Kathy's advisor in the single transaction (89). Otherwise, definitions (87) and (88) would violate the fundamental law of semantic consistency.

7.8 Binary Relations

The binary relation, teach, between faculty members and courses can be defined as:

(90) $\vdash$ teach == ()x:#Faculty)()y:#Course)T Accept

Any faculty member can teach any course.

When a particular teaching assignment is made, the following extensional definition can be entered:

(91) $\vdash$ teach := ()x)()y)(x=Cox $\rightarrow$ y=CS360;
x=Gillett $\rightarrow$ (y=CS135 $\vee$ y=CS236);
x=Kimura $\rightarrow$ y=CS301) Accept

Cox teaches CS360, Gillett teaches CS135 and CS236, and so on. The above extensional definitions are consistent with (90) because the following condition is satisfied:

For arbitrary symbols a and b,

a $_*$ (b $_*$ #teach) :=: T implies a . (b . teach) :=: T.

This is the fundamental law of semantic consistency applied to the second-order abstraction name 'teach'.

If a course attribute, instructor, is already defined as

(92) $\vdash$ instructor == ()x:#Course)()/y:#Faculty)T

with appropriate extension, then an alternative to the the binary relation teach, defined in (91), can be derived from the instructor attribute as:

(93) $\vdash$ teach == ()x:#Faculty)()y:#Course)(y_*#instructor = x).

8. Advanced Capabilities

In Section 7 we illustrated the basic capabilities of ADS in representing a view of reality based on entity types and attribute functions. In this section we demonstrate a selection of other ADS capabilities.

8.1 Relational View of Reality

There are two ways of representing an n-ary relation (n>0) in ADS: one is by an n-th order abstraction as shown in Section 7.7 and the other is by a first-order abstraction representing a set of n-tuples. For example, consider the following definition of a second-order abstraction name 'teach':

$\vdash$ teach == ()x:#Faculty)()y:#Course)T.

The abstraction teach is a second-order function from the entity type Faculty to a propositional function which in turn maps Course to the logical values. An alternative way of representing the binary relation is as follows:

$\vdash$ TEACH == ()<x:#Faculty, y:#Course>)T.

This is a first-order abstraction representing a subset of the cartesian product of Faculty and Course. With this definition the following equivalence holds:

For arbitrary symbols a and b,

<a, b>.TEACH :=: b.(a.teach).

The ADS representation of an arbitrary n-ary relation is:

$$\vdash \; R == (\lambda\langle A_1{:}D_1,\ A_2{:}D_2,\ \ldots\ ,\ A_n{:}D_n\rangle)T$$

where the A_i's are abstraction variables (attribute names) and the D_i's are type names (domain names) [11]. An insertion and deletion of an n-tuple $\langle a_1, a_2, \ldots , a_n\rangle$ to and from R can be made by:

$$\vdash \; \langle a_1, a_2, \ldots , a_n\rangle * \#R := T$$

$$\dashv \; \langle a_1, a_2, \ldots , a_n\rangle * \#R := T.$$

If the relation must satisfy certain constraints such as functional dependencies, then the constraints can be incorporated in the intensional definition. For brevity, let n=4, and assume that the following functional dependencies must be satisfied by the extension of R (using the notation of Chapter 5 in [35]):

$$\{\ A_1 \rightarrow A_2,\ A_1A_3 \rightarrow A_4\ \}.$$

The the following ADS definition specifies the required functional dependencies:

$$\vdash \; R == (\lambda\langle A_1{:}D_1,\ A_2{:}D_2,\ A_3{:}D_3,\ A_4{:}D_4\rangle)$$
$$(\forall\langle u,v,w,x\rangle{:}\#R)$$
$$(\ (A_1{=}u \rightarrow A_2{=}v;\ F) \wedge (A_1{=}u \wedge A_3{=}w \rightarrow A_4{=}x;\ F)\ ).$$

Similarly, if the extension of R should be in the third normal form (3NF), then a corresponding constraint can be incorporated into the definition.

In order to prove the relational completeness of the ADS data language, it is sufficient to show that the following operations [35] can be represented in the language: (i) set union, (ii) set difference, (iii) cartesian product, (iv) projection, and (v) selection. Since these operations are representable in the ADS language, ADS is relationally complete.

Similarly, the notions of generalization, aggregation, and classification in the sense of [32] can also be represented in ADS.

8.2 Transitive Closure of Binary Relations

One of the limitations of the relational data model is its inability to represent the transitive closure of a binary relation. In ADS we can construct not only the transitive closure of a specific binary relation but also the general operator (Closure), that takes a descriptor (or a name) of a binary relation as an argument and yields a descriptor for the transitive closure of the binary relation. We illustrate the construction of Closure by constructing the transitive closure of the Prerequisite relation between two courses. For brevity, we use only intensional information.

(94) $\vdash$ level == ()x:Course)(↿ y:Number)(0<y<7)

Any number between 0 and 7 can be the level of a course.

(95) $\vdash$ Prerequisite == ()<x:Course, y:Course>)(x.level < y.level)

Course x is a prerequisite of Course y where the level of x is lower than the level of y.

Let Required be the transitive closure of Prerequisite. Then

(96) $\vdash$ Required == ()<x:Course, y:Course>)

(<x,y>.Prerequisite → T; (∃ z:Course)

(<x,z>.Prerequisite∧<z,y>.Required → T; F)).

Course x is required for course y if x is a prerequisite of y, or x is a prerequisite of some course z, such that z is required for y.

The closure operator can be defined by generalizing the above construction as:

(97) $\vdash$ Closure == (λr:Relation)(λ<x,y>)(<x,y>.[r] $\rightarrow$ T;

($\exists$z)(<x,z>.[r] $\wedge$ <z,y>.(r.Closure) $\rightarrow$ T; F))

where 'Relation' is a name of the syntactic category for descriptors of binary relations on a set. Note that the last occurrence of 'r' in the abstraction body is not bracketed unlike all other occurrences because Closure takes a symbol as an argument. There are no type specifications for the variables 'x', 'y', and 'z' because they depend on the argument 'r'. To see how the operator can be used, consider

(98) $\vdash$ Required == "Prerequisite" . Closure.

Then,

Required

$\equiv$ "Prerequisite".Closure

$\equiv$ (λ<x,y>)(<x,y>.Prerequisite $\rightarrow$ T;

($\exists$z)(<x,z>.Prerequisite$\wedge$<z,y>.("Prerequisite".Closure) $\rightarrow$ T; F))

$\equiv$ (λ<x,y>)(<x,y>.Prerequisite $\rightarrow$ T;

($\exists$z)(<x,z>.Prerequisite$\wedge$<z,y>.Required $\rightarrow$ T; F))

Thus, definition (98) satisfies the condition for being the transitive closure of Prerequisite.

Note that the argument for Closure need not be a name. Any first-order generalization descriptor of the following form can be the argument:

(λ<x:A, y:A>)P(x,y)

where A is a type name and P(x,y) is an arbitrary predicate.

8.3 Fixed Point Operator

The purpose here is to demonstrate that ADS can represent highly abstract concepts and to show that ADS's capability for recursive definition of names is not essential. The results are mainly of theoretical interest. In recursive function theory recursion can be eliminated by the fixed point operator [33]. In the lambda calculus, the fixed point operator is:

fix = [λf.[]λx.f(x(x))][]λx.f(x(x))]].

In ADS it is:

⊢ Fix == ()λf)('()λx)('(x.[x])'.[f])'.()λx)('(x.[x])'.[f])).

Note that the quotes used in the descriptor are all transparent quotes. For an illustration of how Fix can be used, see [10], Appendix I.

9. Conclusions

We are currently exploring the addition of the object categories process and event. The process concept allows us to specify not only the descriptive semantics of data states but also the prescriptive semantics of database manipulation operations. Currently ADS has no control mechanism for sequencing commands and queries.

Inclusion of the process concept also extends the scope of the data model from a single user model to a multi-user model in which the database management system interacts concurrently with multiple users. We are studying the notion of a multidatabase as a design tool for modularization of such a multi-user database system. A multidatabase is a hierarchically structured society of cooperating and independent database managers communicating with each other by exchanging messages. The concept of process (and multidatabase) is a natural extension of the concept of module in programming languages such as ADA.

We are also considering the addition of event as an ADS object category to represent temporal precedence relationships. As mentioned previously, an elementary datum in [20] is a 4-tuple

<object name, object property, property value, time>,

where the first three components correspond to an ADS database object. An important next step in the process of completing our data model is to decide how to incorporate the time component into the ADS model. Generalizations on events called "occurrence counting functions" have been proposed in [9] as a specification mechanism for controlling event occurrences in a concurrent system such as a multidatabase.

An implementation of the full ADS model requires the realization of deduction capabilities for logical consistency checking among intensional data. Such capabilities require significant computational resources. By the same token, pattern matching (searching) and pattern association (binding) operations that are applied to a large extensional database also demand heavy usage of computational resources and extensive memory resources.

The C implementation simulates some of these capabilities and restricts some forms of the ADS syntax so that a reasonable performance level can be maintained. For example, in some contexts a restriction is applied that variables can only range over extensional database values.

We have successfully tested the feasibility of ADS with an implementation in SAIL on TOPS-20. We have started a prototype production implementation in C on UNIX for a Motorola 68000 system.

Microelectronic technology, supported by the proper design of concurrent algorithms for pattern manipulation operations, has a great potential for satisfying the need for additional computational resources. Our current efforts in this direction concentrate on the development of custom LSI chips that integrate distributed memory space for binary trees (ADS symbols) with highly concurrent pattern matching, binding, and substitution capabilities. Testing is currently under way on an experimental chip for an associative memory subsystem, fabricated using the University of Southern California Information Sciences Institute's MOS Implementation System (MOSIS).

In parallel to the development of hardware support for a full implementation of ADS, a development of efficient resolution algorithms and unification algorithms based on binary tree pattern manipulation operations is being pursued.

In summary, some specific advantages of an ADS database are:

(1) Well defined semantic consistency.

(2) Arbitrary levels of abstraction.

(3) Ability to model an intensional world as well as an extensional world with flexible interaction between them.

(4) Ability to update and query database schemata using the same data language as used for the object database.

(5) Powerful specification capability of high-level concepts such as the general transitive closure operation and the fixed point operator.

References

[1] Abrial, J. R. Data semantics. In Data Base Management, J. W. Kimble (Ed.), Amsterdam: North-Holland, 1974, 1-60.

[2] Carnap, R. Meaning and Necessity, 2d ed., Chicago, Illinois: University of Chicago Press, 1956.

[3] Carroll, J. M. Toward an integrated study of creative naming. Research Report RC9016(#39483), IBM Watson Research Center, Yorktown Heights, 1981.

[4] Chen, P. P. The entity-relationship model: Towards a unified view of data. ACM Transactions on Database Systems 1,1 (Mar. 1976), 9-36.

[5] Church, A. The Calculi of Lambda Conversions. Princeton, New Jersey: Princeton University Press, 1941.

[6] Church, A. Carnap's introduction to semantics. The Philosophical Review. 52 1943, 298-304.

[7] Church, A. Introduction to Mathematical Logic. Princeton, New Jersey: Princeton University Press, 1956.

[8] Codd, E. F. Extending the database relational model to capture more meaning. ACM Transactions on Database Systems. 4:4 (Dec. 1979), 397-434.

[9] Cox, Jr., J. R. A medical information system design methodology. Annual Report to National Center for Health Services Research, Department of Computer Science, Washington University, St. Louis, Missouri, (June 1980).

[10] Cox, Jr., J. R. A medical information system design methodology. Annual Report to National Center for Health Services Research, Department of Computer Science, Washington University, St. Louis, Missouri, (June 1982).

[11] Date, C. J. An Introduction to Database Systems, 2d ed., Reading, Massachusetts: Addison-Wesley, 1977.

[12] Enderton, H. B. A Mathematical Introduction to Logic, New York: Academic Press, 1972.

[13] Frege, G. Uber Sinn und Bedeutung. Zeitschrift fur Philosophie und Philosophische Kritik 100, 25-50. Translated by M. Black as "On Sense and Reference" In P. Geach and M. Black. Translations from the Philosophical Writings of Gottlob Frege, Oxford, 1952.

[14] Gallaire, H., J. Minker and J. M. Nicolas. An overview and introduction to logic and data bases. In Logic and Data Base, Eds. H. Gallaire and J. Minker, New York: Plenum Press, 1978, 3-30.

[15] Gorn, S. The identification of the computer and information sciences:

Their fundamental semiotic concepts and relationships. In Foundation of Language, 4 (1968), 339-72.

[16] Hoare, C. A. R. Notes on data structuring. In Structured Programming. New York: Academic Press, 1972, 83-174.

[17] Kent, W. Data and Reality. Amsterdam: North-Holland, 1978.

[18] Kimura, T. D. and W. D. Gillett. Formal specification of ADS, an abstract database system. Technical Report WUCS-81-3, Department of Computer Science, Washington University, St. Louis, Missouri, May 1981.

[19] Kimura, T. D. Semantic abstraction and the concept of type. Technical Report WUCS-82-10, Department of Computer Science, Washington University, St. Louis, Missouri, 1982.

[20] Langefors, B. Information systems theory. Information Systems. 2, 1977, 207-19.

[21] Lyons, J. Semantics, Volume 1. Cambridge University Press, 1977.

[22] McCarthy, J. LISP 1.5 Programmer's Manual. 2d ed., Cambridge, Massachusetts: MIT press, 1965.

[23] McCarthy, J. First order theories of individual concepts and propositions. In Machine Intelligence 9, Edinburgh University Press, 1979, 129-47.

[24] Mill, J. S. A System of Logic. London: Longmans, 1843.

[25] Minker, J. An Experimental Relational Data Base System Based on Logic. In Logic and Data Base, Eds. H. Gallaire and J. Minker, New York: Plenum Press, 1978, 107-47.

[26] Mylopoulos, J., P. A. Bernstein and H. K. T. Wong. A language facility for designing database-intensive applications. ACM Transactions on Database Systems. 5:2 (June 1980), 185-207.

[27] Perlis, D. R. Language, computation, and reality. Ph.D. Thesis, Department of Computer Science, University of Rochester, 1981.

[28] Quine, W. V. Methods of Logic. New York: Holt, Rinehart and Winston, 1950.

[29] Russell, B. Mathematical Logic as Based on the Theory of Types. Americal Journal of Mathematics. 30, 1908, 222-62.

[30] Russell, B. The philosophy of logical atomism. In Logic and Knowledge., Ed. R. C. Marsh, New York: Putnum's Sons, 1956, 177-281.

[31] Shipman, D. W. The functional data model and the data language DAPLEX. ACM Transactions on Database Systems. 6:1 (Mar. 1981), 140-73.

[32] Smith, J. M. and D. C. P. Smith. Database abstractions: Aggregation and generalization. ACM Transactions on Database Systems.

2:2 (June 1977), 105-33.

[33] Stoy, J. Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, 1977.

[34] Tsichritzis, D. C. and F. H. Lochovsky. Data Models. Englewood Cliffs, New Jersey: Prentice-Hall, 1982.

[35] Ullman, J. D. Principles of Database Systems. Computer Science Press, 1980.

[36] Yap, C. K. A semantic analysis of intensional logic. Technical Report RC6893 (#29538), IBM Thomas J. Watson Research Center, Yorktown Heights, New York, December 1977.