A Design of a Data Model Based on Abstraction of Symbols

T.D. KIMURA, W.D. GILLETT, AND J.R. COX, Jr.
Department of Computer Science, Washington University, St. Louis, Missouri 63130, *USA*

Abstract Database System (ADS) is introduced as 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. 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.

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[8,9,16].

A set of conceptual tools organized for representing a user's knowledge about reality is called a *data model*[25]. Different data models utilize different sets of conceptual tools. The relational data model[7] 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[23] uses the concepts of set and multi-valued function. A binary data model (eg, in reference 1) uses the concepts of category (set) and access function (binary relation). The semantic network data model in TAXIS[19] uses the concepts of class, property, and generalization (IS-A relationship). The logical data model in MRPPS[18] uses the calculus of many sorted logic[10].

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 (eg, the lambda calculus of reference 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.

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 raw data.
(4) Abstractions on Symbols: Users can name and describe generalizations 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.

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 Institute of Health.

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.

2.1 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[11] 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 following two symbols (expressions):

(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.

Frege associates a meaning other than denotation, with (1) and (2), which he calls *sense*.

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 '≡' 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".

2.2 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'.

In the database literature[12], 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)	TEACHING (instructor, course)
Cox	Kimura CS135
Gillett	Gillett CS236
Kimura	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.

2.3 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).

2.4 Unquote

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 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 reference 20 within the framework of the first order predicate calculus).

Consider for example the following symbols:

(5) "The most common name" denotes "John"
(6) [The most common name] is a student
(7) ["John"] is a student
(8) [John] is a student.

Statement (5) establishes a semantic relationship between two symbols; the first phrase 'the most common name' is a symbol that denotes another symbol 'John'. Statements (6) and (7) are equivalent to 'John is a student', because the bracketed expression in (6) '[The most common name]' and that in (7) '["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 (8) 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. Note that in this paper, we differentiate between quotes in data (") and quotes in English (' ').

2.5 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.

2.6 Abstractions on Symbols

Smith and Smith[24] 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 in which abstraction is independent of the symbolism used for representing it. Our theory is a theory of symbols in which abstraction is possible only through symbolism.

2.6.1 *Generalization*

In ADS, the mechanism of abstraction, called *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.

Consider the following statements:

(9) John is a student
(10) Dave is a student.

These two statements (symbols) have the same form as

(11) ____ is a student,

where the blank can be filled with a symbol denoting a person. This form displays the similarity between (9) and (10), i.e., they share the symbol 'is a student'. The form can be interpreted as an abstraction of many symbols similar to (9) and (10). This interpretation of (11), generalization, is represented in ADS by (12),

(12) (λx:Person)(x is a student).

The symbol 'x', called the *abstraction variable*, plays the role of a place holder corresponding to the blank in (11). The common symbol 'x is a student' is the *abstraction body*.

Symbol (12) is a *generalization descriptor* or an *abstraction descriptor*. 'x:Person' indicates that x must be of type Person (ref. 2.7). ADS abstractions are primarily used to represent sets, functions and types. For example, abstraction (11) 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.

2.6.2 *Instantiation*

The inverse of abstraction is called *instantiation*. If abstraction (12) is the result of generalization on (9) and (10), then the instantiation of (12) yields (9) when based on the symbol 'John' and (10) when based on the symbol 'Dave'.

We differentiate between *intensional* instantiation and *extensional* instantiation based on the synonymy/equivalence distinction. An intensional instantiation is represensed by the dot notation:

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

Here the symbol specified by the expression preceding the dot in (13), 'John', is the *basis* of the instantiation, and the result of (13) is 'John is a student'.

Recall that two symbols are synonymous (≡) if they express the same sense. We define (13) to be synonymous with (9). Thus,

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

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

(14) "John" * (λx:Person)(x is a student)

Instantiation (14) is synonymous, by definition, with a symbol that is equivalent to (having the same denotation as) (9). Since there are many symbols equivalent to (9), (e.g. (10)), (14) 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".

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,

$$a \,.\, (\lambda x)A(x) \equiv b \qquad \text{iff } A(a) \equiv b,$$

and

$$a * (\lambda x)A(x) \equiv b \qquad \text{iff } A(a) :=: b.$$

2.7 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 (9) and (10) are abstracted (generalized) into (12), it is intended to be a generalization of more than just (9) and (10). 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 (12), therefore, must be within the intended range of abstraction. Thus,

"CS135" . (λx: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 (12).

Note that the above concept of type is akin to the one used in logic, particularly in the theory of types[21], where logically paradoxical symbols are discarded as nonsignificant due to type violation. However, it is different from common notions of type in programming language and database literature (eg, in reference 13), where a type is defined as a category of objects. Russell[22] 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 in reference 15.

2.8 Transparent Quote

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

(15) "John" is the name of John
(16) "Dave" is the name of Dave.

If we try to generalize the above symbols into

(17) (λx: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 (17) based on 'John' will yield '"x" is the name of John', instead of (15). With the transparent quote, the proper generalization of (15) and (16) is

(λx:Person)(|x| is the name of x)

whose intensional instantiations will yield (15) and (16).

2.9 Description Operator

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

($\imath$x)P(x) $\equiv$ a iff P(a) :=: "Truth"
for an arbitrary symbol a.

Thus, '($\imath$x)P(x)' means '*an* x such that P(x)', and it is nonsignificant only when P(x) is true of nothing. We call the iota operator the *description operator*.

3. DATABASE USE OF SYMBOLS

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

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 a sentence.

3.1 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 (eg, '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 (eg, '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[6] and Carnap[2].

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*.

3.2 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:

(18) ABC == the best student in CS135
(19) DEF := the best student in CS135.

Definition (18) defines 'ABC' as a name expressing the same sense that the descriptor expresses, and (19) 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 (19) 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 (18) is an *intensional definition* of the name 'ABC' where the sense (intension) of the descriptor defines the intension of the name. Statement (19) 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 '@'.

3.3 Intensional and Extensional Use of Names

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.

(20) ABC == the best student in CS135
(21) ABC := John.

Then, the statement

(22) ABC takes EE280

is ambiguous as to whether it is synonymous with (23) or only equivalent to (24):

(23) The best student in CS135 takes EE280
(24) John takes EE280.

When (22) is intended to mean (23), we say the name 'ABC' is *intensionally used*. When it is intended to mean (24), the name is *extensionally used*.

In order to eliminate such ambiguity, an 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 (20) and (21), respectively:

"ABC takes EE280" ≡ "The best student in CS135 takes EE280", and
"#C takes EE280" :=: "John takes EE280".

3.4 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 called 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, the 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.

3.5 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, a statement is called a *boolean desoriptor*. For example,

(25) John is a student

expresses the proposition that John is a student and denotes Truth if John actually is a student.

In ADS, there are two different ways of representing statement (25): 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 (25) 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 (25) by its (intensional) instantiation as:

(26) "John" . Student.

If the abstraction Student is interpreted as a certain class of person, then (26) expresses set membership. If it is interpreted as a predicate function, then (26) expresses a function application.

The other way is to consider that it 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 = = (x:Person)(x has paid tuition).

Then we can represent (25) by:

John = student,

where in ADS a = b iff a :=: b by definition.

Thus, in ADS, a statement (boolean descriptor) is represented either with '=' or '.', or a logical composition of these statements with the standard logical operators, '$\wedge$' (and), '$\vee$' (or), '$\sim$' (not), '$\forall$' (for all), and '$\exists$' (for some).

3.6 Assertion of Statement

Database users update a database by either *asserting* a statement or by *de-asserting* a statement which had been previously asserted (ie, cancelling a previous assertion). The logical assertion symbol '$\vdash$' indicates an assertion, and the symbol '$\dashv$' indicates a de-assertion as in:

(27) $\vdash$ John is a student
(28) $\dashv$ John is a student.

Note that a de-assertion is different from a negative assertion. Deassertions decrease the amount of information obtainable from a database, while negative assertions increase it. After the judgment represented by (28) is made, it is not known whether John is a student or not. After entering the following negative assertion,

(29) $\vdash$ John is not a student

it is known that John is not a student. Assertion (29) is inconsistent with (27), but is consistent with (28).

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

$\vdash$ <name> = = <descriptor>
or $\vdash$ <name> := <descriptor>.

Here '<name>' represents an arbitrary name and '<descriptor>' represents an arbitrary descriptor. Therefore, in order to enter assertion (27), a user would decompose it into two parts; first, name the proposition then define the extension of the name to be true:

(30) $\vdash$ FACT = = John is a student
(31) $\vdash$ FACT := Truth.

3.7 Constraints and Transactions

When a boolean name is extensionally defined (e.g., (31)) the database system first checks the semantic consistency with the intensional definition (e.g., (30)), 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 names for logical values 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.

4. 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. If the command is an update command, then it 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.

The ADS database state consists of a set of symbols 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[17]:

(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

$\vdash$ 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

$\vdash$ 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'.

5. SCHOOL DATABASE

In this section, using a school database as an example, we illustrate the ability of ADS to integrate an intensional database with an extensional one in a flexible manner 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 a variety of ADS language constructs, we view the world in terms of entity types and attribute functions as in reference 23, rather than a set of relations.

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 '$\equiv$' 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 reference 14) 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'.

5.1 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:

(32) $\vdash$ Person = = (λx)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.

(33) ? 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

Cox.Person $\equiv$ Cox.(λx)T $\equiv$ 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, ie, '# Person' is nonsignificant, which corresponds to the fact that no person is actually known yet. Therefore,

(34) ? Cox $\in$ # 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:

(35) Person := (λx)(x=Cox → T) Accept
Cox is a person.

(36) ? Cox * #Person Yes
Is Cox a person?

Definition (35) associates the extension of the name 'Person' with the denotation of the descriptor '(x)(x=Cox → T)'. The response to (36) is 'Yes' because

Cox * #Person
:=: Cox * (λx)(x=Cox → T)
:=: (Cox=Cox → T)
:=: T.

Extensional definition (35) is accepted by the database because it is consistent with the intensional definition (32) 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 *is* and *can be*). This is the definition of the fundamental law of semantic consistency applied to the abstraction name 'Person'.

5.2 Update of Entity Types

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

(37) ⊢ Person :=
(λx)(x=Gillett → T; x * #Person) Accept
Gillett is also a person

(38) ? Cox * #Person Yes
Is Cox a person?

Note that in (37) the new extension of 'Person' is defined in terms of the old extension of the name. Since prior to (37),

#Person :=: (λx)(x=Cox → T),

the response of (38) 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.

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

(39) ⊢ Course = = (λx)T

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

(41) ⊢ Student = = (λx: Person)T

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

(43) ⊢ Faculty = = (λx: Person)T

(44) ⊢ Faculty :=
(λx: Person)(x=Cox ∨ x=Gillettx=Kimura)

5.3 Consistency Checking

We show here that a local update may involve a global consistency check. Definitions (32) and (39) 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 (39):

(45) ⊢ Course = = (λx)(x * #Person → F; T) Accept

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

Even though definition (45) explicitly restricts only 'Course', it also implicitly restricts 'Person' as well. Consider the following update assertion attempted after (32), (45) and (40) have been entered:

(46) 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. 3.7):

(47) <
⊣ Person = = (λx)T
⊢ Person = = (x)(x * #Course → F; T)
> Accept.

5.4 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. 3.7) on the extensions of 'Person' and 'Course':

(48) ⊢ Course-Constr = =
(∀x)(x * #Person → ~(x * #course); T) Accept
For any object, if it is a person, then it is not a course.

(49) ⊢ Course-Constr := T Accept
Let Course-Constr be true.

Assertion (49) is accepted by the database because after it is entered the database state is consistent with (48) 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 (49) 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.

5.5 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:

(50) ⊢ advisor = =
(λx: Student)(ιy: # Faculty)(~x=y) Accept
Any faculty member can be the advisor of any student but himself.

Note that from (41) and (43) 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:

(51) ? Dave.advisor Cox (Gillett, Kimura)
Who can be Dave's advisor?

The above query is synonymous to

? (ιy: # 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:

(52) ⊢ advisor :=
(λx)(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 (50) 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'.

5.6 Binary Relations

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

(53) ⊢ teach = =
(λx: # Faculty)(λy: # Course)T Accept
Any faculty member can teach any course.

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

(54) ⊢ teach := (λx)(λy)(x=Cox → y=CS360;
x=Gillett → (y=CS135 ∨ y=CS236);
x=Kimura → y=CS301) Accept
Cox teaches CS360, Gillett teaches CS135 and CS236, and so on.

The above extensional definitions are consistent with (53) 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'.

6. ADVANCED CAPABILITIES

In order to prove the relational completeness of the ADS data language, it is sufficient to show that the following operations can be represented in the language: (i) set union, (ii) set difference, (iii) cartesian product, (iv) projection, and (v) selection[26]. Since these operations are representable in the ADS language, ADS is relationally complete. 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.

(55) ⊢ level = = (λx:Course)(ιy:Number)(0<y<7)
Any number between 0 and 7 can be the level of a course.

(56) ⊢ Prerequisite = =
(λ<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

(57) ⊢ Required = = (λ<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:

(58) ⊢ Closure = =
(λr:Relation)(λ<x,y>)(<x,y >.[r]→ T;
(∃z)(<x,z>.[r] ∧ <z,y>.(r.Closure) → 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

(59) ⊢ Required = = "Prerequisite" . Closure.

Then,

Required
≡ "Prerequisite".Closure
≡ (λ <x,y>)(<x,y>.Prerequisite → T;
(∃z)(<x,z>. Prerequisite ∧
< z,y>.("Prerequisite".Closure)
→ T; F))
≡ (λ <x,y>)(<x,y>.Prerequisite → T;
(∃z)(<x,z>.Prerequisite ∧ < z,y>
.Required → T; F))

Thus, definition (59) 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. Other highly abstract concepts can be represented in ADS. For example, the fixed point operator of lambda calculus is easily represented and used[16].

7. CONCLUSIONS

We have successfully tested the feasibility of ADS with a prototype production implementation written in the language C. 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.

An implementation of the full ADS model requires the realization of deduction capabilities for logical consistency checking between 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.

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, Stuart Goldkind, Pat Moore, Bill Ball and other members of the Information Systems Group who have been good listeners and have provided various forms of support. Finally, the authors appreciate the valuable comments provided by one reviewer which improved the readability of the paper.

REFERENCES

1. J.R. Abrial, Data Semantics. In *Data Base Management*. J. W. Kimble (Ed.), Amsterdam, North-Holland, 1–60, (1974).
2. R. Carnap, *Meaning and Necessity*, 2d ed., Chicago, Illinois: University of Chicago Press, (1956).
3. J.M. Carroll, Toward an Integrated Study of Creative Naming. Research Report RC9016(39483), IBM Watson Research Center, Yorktown Heights, (1981).
4. P.P. Chen, The Entity-Relationship Model: Towards a Unified View of Data. *ACM Transactions on Database Systems* **1,1**, (Mar. 1976), 9 – 36.
5. A. Church, *The Calculi of Lambda Conversions*. Princeton, New Jersey: Princeton University Press, (1941).
6. A. Church, *Introduction to Mathematical Logic*. Princeton, New Jersey: Princeton University Press, (1956).
7. E.F. Codd, Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*. **4:4** 397–434, (Dec. 1979).
8. J.R. Cox, Jr., *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).
9. J.R. Cox, Jr., *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).
10. H.B. Enderton, *A Mathematical Introduction to Logic*. New York, Academic Press, (1972).
11. G. Frege, 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).
12. H. Gallaire, 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, 3–30, (1978).
13. C.A.R. Hoare, Notes on Data Structuring. In *Structured Programming*. New York: Academic Press, 83–174, (1972).
14. W. Kent, *Data and Reality*. Amsterdam, North-Holland, (1978).
15. T.D. Kimura, Semantic Abstraction and the Concept of Type. Technical Report WUCS–82–10, Department of Computer Science, Washington University, St. Louis, Missouri, (1982).
16. T.D.,Kimura, W.D. Gillett and J.R. Cox, Jr. Abstract Database System (ADS): A Data Model Based on Abstraction of Symbols. Technical Report WUCS–82–12, Department of Computer Science, Washington University, St. Louis, Missouri, (July, 1982).
17. B. Langefors, Information Systems Theory. *Information Systems*. **2**, 207–19, (1977).
18. J. Minker, An Experimental Relational Data Base System Based on logic. In *Logic and Data Base*. Eds. H. Gallaire and J. Minker, New York, Plenum Press, 107–47, (1978).
19. J., Mylopoulos, P.A. Bernstein and H.K.T. Wong. A Language Facility for Designing Database-Intensive Applications. *ACM Transactions on Database Systems*. **5:2**, 185–207, (June 1980).
20. D.R. Perlis, *Language, Computation, and Reality*. Ph.D. Thesis, Department of Computer Science, University of Rochester, (1981).
21. B. Russell, Mathematical Logic as Based on the Theory of Types. *Americal Journal of Mathematics*. **30**, 222–62, (1908).
22. B. Russell, The Philosophy of Logical Atomism. In *Logic and Knowledge*, Ed. R.C. Marsh, New York: Putnum's Sons, 177–281, (1956).
23. D.W. Shipman, The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*. **6:1**, 140–73, (March 1981).
24. J.M. Smith, and D.C.P. Smith. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems*. **2:2**, 105–33, (June 1977).
25. D.C. Tsichritzis and F.H. Lochovsky, *Data Models*. Englewood Cliffs, New Jersey: Prentice-Hall, (1982).
26. J.D. Ullman, *Principles of Database Systems*. Computer Science Press, (1980).