Report Number: WUCS-83-5

1983-12-01

ADS Formal Semantics

Takayuki Kimura

Abstract Database System (ADS) is a data model developed for an enduring medical information system where frequent changes in the conceptual schema are anticipated and multi-level abstraction is required. The mechanism of abstraction in ADS is based on the abstraction operator of the lamba calculus. The formal semantics of a subset of the ADS model is presented using the denotational specification method.

ADS FORMAL SEMANTICS

Takayuki Kimura

WUCS-83-5

DEPARTMENT OF COMPUTER SCIENCE
WASHINGTON UNIVERSITY
ST. LOUIS, MISSOURI 63130

December 1983

ABSTRACT

Abstract Database System (ADS) is a data model developed for an enduring medical information system where frequent changes in the conceptual schema are anticipated and multi-level abstraction is required. The mechanism of abstraction in ADS is based on the abstraction operator of the lambda calculus.

The formal semantics of a subset of the ADS model is presented using the denotational specification method.

ADS Formal Semantics

1. Introduction

This appendix specifies the formal semantics of a subset of the ADS data model. This specification demonstrates that the semantics of the entire data model can be formally defined. The denotational specification method has been chosen over the axiomatic and operational methods because of its simplicity and elegance [3]. We assume that the reader is familiar with both the ADS data model and the denotational method of semantic specification.

We consider a subset of ADS that is simple to understand, yet still contains the essential, fundamental parts of ADS. For example, because the existential quantifier ($\exists$) can be represented by a combination of logical negation and the universal quantifier($\forall$), only the universal quantifier is included in the subset. Similarly, the concept of type is not included in the subset because the typing of abstraction variables can be represented by a combination of a conditional descriptor and a test of a symbol's significance (i.e., whether a symbol's meaning has been defined or not).

Section 2 defines the syntax of the ADS subset. The syntax of the syntactic classes (descriptors, transformers, predicates, updates and queries) are specified. Section 3 discusses some of the key ADS concepts. Also, the notation used to define the semantic evaluation function is introduced. Section 4 discusses name evaluation which lies at the very core of the semantics of ADS. Section 5 formally defines semantic consistency. The relationship between intensional and extensional descriptions is discussed. Section 6 defines update evaluation. Finally, Section 7 defines query evaluation.

2. ADS Syntax

For the remainder of this appendix, we will use the term ADS to mean the language system whose syntax is defined in this section and whose semantics are defined in the following sections. We will use a BNF-like notation to specify the syntax using "::=" for defining syntactic categories and "|" for defining an alternative construction. Higher order functions (transformers) are represented by:

$$t_0, t_1, t_2, \dots$$

where t_i is the category of i-th order transformers. Similarly, higher order predicates are represented by:

$$p_0, p_1, p_2, \dots$$

where p_i is the category of i-th order predicates.

Furthermore, we leave the syntactic category of names (identifiers) undefined. The symbol "n" denotes a name from the set of arbitrary names which includes the terminal symbols "N", "T", and "F".

The terminal symbols are:

+	'	"	(	)	/	ι	λ	→	⊢	∀	#
.	;	[	]	T	F	N	=	==	:=	?	@

For the sake of readability, terminal symbols are not quoted in the syntax specification. A brief explanation of these symbols is given in Section 3, Appendix I of this report. With the following abbreviations for syntactic categories:

n : name,

d : descriptor,

t_i : i-th order transformer, $(i \geq 0)$

p_i : i-th order predicate, $(i \geq 0)$

u : update,

q : query,

the syntax of ADS can be defined as follows:

(1)	d	::=	$t_i \mid p_i \quad (i \geq 0)$
(2)	t_0	::=	$N \mid +t_0t_0 \mid \text{'d'} \mid \text{"d"} \mid @n \mid (\iota n)p_0$
	t_i	::=	$t_0 \cdot t_{i+1} \mid p_0 \rightarrow t_i;\ t_i \mid n \mid \#n \mid [t_0]$
	t_{i+1}	::=	$(\lambda n)t_i$
(3)	p_0	::=	$T \mid F \mid /t_0/ \mid t_0=t_0 \mid (\forall n)p_0$
	p_i	::=	$t_0 \cdot p_{i+1} \mid p_0 \rightarrow p_i;\ p_i \mid n \mid \#n \mid [t_0]$
	p_{i+1}	::=	$(\lambda n)p_i$
(4)	u	::=	$\vdash n{=}{=}d \mid \vdash n{:}{=}d$
(5)	q	::=	?d

Although the language described by the above definition is only a subset of the full ADS language, it contains enough variety so that the basic conceptual foundation upon which the ADS model rests is exemplified in the semantic definitions which follow. The motivation for the syntactic categories and the semantics which follow is elaborated upon in Reference 1. We now turn to a discussion of the fundamental concepts required to understand the evaluation function in ADS.

3. Preliminaries

In this section, we discuss some basic concepts and introduce the notation used to define the semantic evaluation function. These concepts include the ADS encoding function, ADS objects, an ADS database and the symbolic operation of free substitution. Standard mathematical notation (e.g., for set and function) is used in our formal definitions. For example, "$A \rightarrow B$" denotes the set of all functions from A to B, and "==" is synonymous with "is identical to by definition."

3.1 Encoding Function

An ADS database is a collection of descriptors and names stored in the form of binary trees (defined formally in the next section). An encoding function "f" determines a unique encoded descriptor in the set D from each descriptor from the set Desc representing all possible descriptors in the ADS language. A similar encoding function determines encoded names from unencoded names. More formally,

$$f : \mathrm{Desc} \rightarrow D \qquad D \subseteq B$$

$$f : \mathrm{Name} \rightarrow \mathrm{Na} \qquad \mathrm{Na} \subseteq B$$

where B is the set of binary trees, the encoding form used in ADS.

For example, the 0th order transformer "N" in Section 2 is an unencoded name. It is encoded as a binary tree, as are the other 0th order transformers such as "@name" where "@" is a terminal symbol and "name" is some unencoded name. The 0th order predicate "T" is also a name and is also encoded as a binary tree. The descriptor "(λx) (x.#name=T)" which describes the set of all binary trees which are in the extension of "name" (i.e., #name) is also stored as a binary tree, as is the subdescriptor "x.#name=T".

3.2 Objects

There are four categories of objects that can be described and named in ADS: logical values, symbols, transformations on symbols, and conditions on symbols. ADS symbols are binary trees. A binary tree is denoted by a sequence of ones for internal nodes and zeros for leaf nodes where the order is a preorder traversal. The ADS objects are denoted by the following symbols:

$2 == \{\text{true}, \text{false}\}$	logical values
$B == \{0, 100, 10100, 11000, \ldots\}$	symbols (binary trees)
$Z == \bigcup_{k=0}^{\infty} (T_k \cup P_k)$	objects
where	
$T_0 == B, \quad T_{i+1} == B \rightarrow T_i$	transformations
$P_0 == 2, \quad P_{i+1} == B \rightarrow P_i$	conditions

Objects result from the evaluation of names and descriptors.

3.3 Database

A database state (or simply a database) in ADS is a collection of names and descriptors. One pair of descriptors is associated with each name: an intensional descriptor and an extensional descriptor. An intensional descriptor denotes what the named object can be, and the extensional descriptor denotes what the named object is (Figure 1).

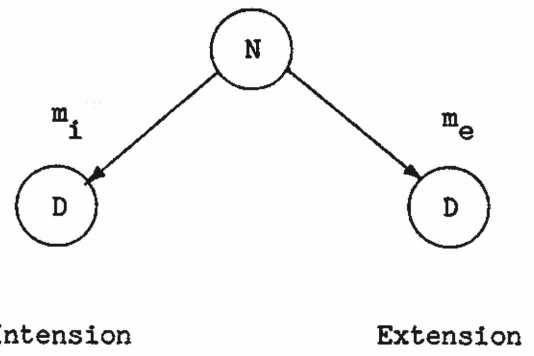

Figure 1. Names and Descriptors.

Formally, a database is an association (mapping) between names and pairs of descriptors. If we denote a database state by "m" and the set of all possible database states by "M", then,

$M == Na \rightarrow D \times D$	database
$m(n) == (m_i(n), m_e(n))$,	where $m \in M$, $n \in Na$,
and $m_i(n) \subseteq D$	intensional descriptor of n
$m_e(n) \subseteq D$	extensional descriptor of n.

The expression $(m_i(n), m_e(n))$ is the descriptor pair associated with the name n in database state m.

3.4 Free Substitution

An occurrence of a name is bound in a symbol (binary tree) if and only if 1) it appears immediately after an operator $\forall$, λ or ι, 2) it appears within the scope (as in lambda calculus) of the operator in the symbol, or 3) it is quoted by ". An occurrence is free if it is not bound. Note that there two quote symbols in ADS: single (') and double ("). Symbols quoted with single quotes remain free. Free substitution is the replacement of all free occurrences of a name within a symbol by another symbol. It plays a fundamental role in the process of descriptor evaluation. The free substitution operation is denoted by { }. Formally, we have

$\{\ \} : D \rightarrow (Na \times D \rightarrow D)$	free substitution
$\{d\}^{n}_{a} == \{\ \}(d)(n,a)$	substitution of $a \in D$ for $n \in Na$ in $d \in D$.

$\{d\}^{n}_{a}$ is a particular substitution applicable to any descriptor. This view can not be represented by the other notation. The notation $\{d\}^{n}_{a}$ is similar to the notation used in [1] and means that every free occurrence of the name n in the descriptor d is replaced by the descriptor a.

4. Descriptor Evaluation

The evaluation of names forms the nucleus of the semantic definition of ADS. Since the association between a name and what the name refers to (Figure 2) is defined in terms of descriptors, the evaluation of names also requires the evaluation of descriptors; i.e., a descriptor must be associated with the referenced object. The descriptor evaluation function for the descriptor d associated with the name n must also be a function of the current database state m because d may be defined in terms of other names whose descriptors may change from one database state to another.

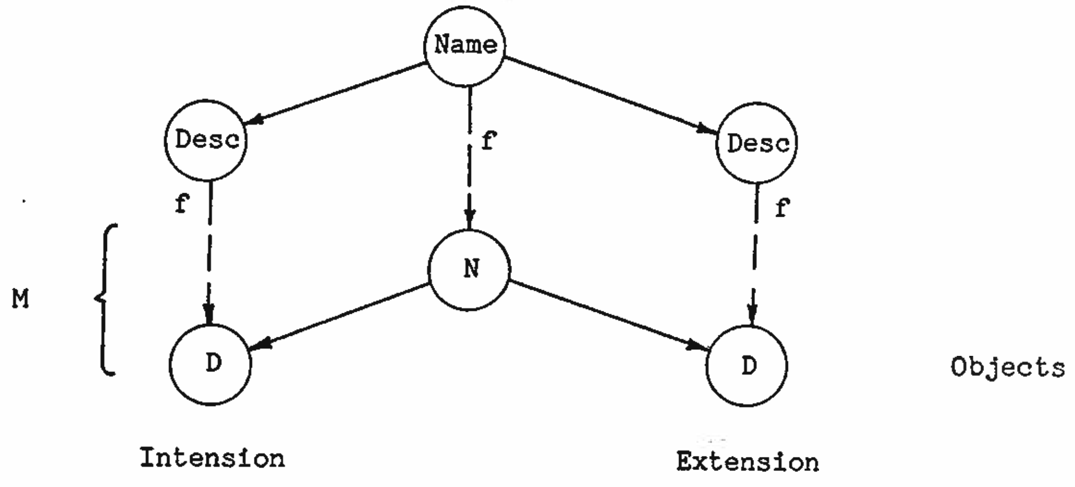

Figure 2. Descriptor Evaluation.

We denote the evaluation function by "[ ]". Then,

$[\;] : D \rightarrow (M \rightarrow Z)$	descriptor evaluation
$[d]_m == [\;](d)(m)$	the meaning of $d \in D$ in database state $m \in M$.

The evaluation function is a partial function on D. We will use the following notational convention:

$$[d]_m == \emptyset \quad \text{iff} \quad [\;] \text{ is undefined on } d \in D \text{ under } m \in M.$$

That is, the evaluation of descriptor d in database state m is empty if and only if the evaluation of d is undefined in database state m.

Below is a recursive definition of the evaluation function [ ] based on the syntactic structure of descriptors as defined in (1), (2) and (3) in Section 2. Bear in mind that all arguments to the evaluation function are encoded symbols (binary trees). However, in order to simplify the notation, the unencoded form of the symbols are used in the definitions. For example, to be precise, in (1) the symbol N should be written as f(N). In the definitions below, t_0, p_0, d and n are syntactic categories. ' $\equiv$ ' denotes the identity on binary trees.

(1)	$\llbracket N \rrbracket == 0$		
(2)	$\llbracket +ab \rrbracket == 1\llbracket a \rrbracket\llbracket b \rrbracket$		$a,b \in t_0$
(3)	$\llbracket 'x' \rrbracket == x$		$x \in d$
(4)	$\llbracket "x" \rrbracket == x$		$x \in d$
(5)	$\llbracket @x \rrbracket_m == m_i(x)$		$x \in n$
(6)	$\llbracket (\iota y)p \rrbracket == x$ such that $\llbracket (\lambda y)p \rrbracket(x) = \text{true}$,		$x \in B,\ p \in p_0,\ y \in n$
(7)	$\llbracket T \rrbracket ==$ true		
(8)	$\llbracket F \rrbracket ==$ false		
(9)	$\llbracket a=b \rrbracket ==$ true if $(\llbracket a \rrbracket \doteq \llbracket b \rrbracket)$ else false		$a,b \in t_0$
(10)	$\llbracket /x/ \rrbracket == (\llbracket x \rrbracket \neq \emptyset)$		$x \in d$
(11)	$\llbracket (\forall y)p \rrbracket == (\forall x \in B)\llbracket (\lambda y)p \rrbracket(x)$		$p \in p_0,\ y \in n$
(12)	$\llbracket (\lambda y)b \rrbracket(x) == \llbracket \{b\}^y_a \rrbracket$ where $\llbracket a \rrbracket = x$		$x \in B,\ b \in d,\ y \in n$
(13)	$\llbracket a.x \rrbracket == \llbracket x \rrbracket(\llbracket a \rrbracket)$		$a \in t,\ x \in d$
(14)	$\llbracket x \rrbracket_m == \llbracket m_i(x) \rrbracket_m$		$x \in n$
(15)	$\llbracket \#x \rrbracket_m == \llbracket m_e(x) \rrbracket_m$		$x \in n$
(16)	$\llbracket\ [a]\ \rrbracket_m == \llbracket \llbracket a \rrbracket_m \rrbracket_m$		$a,\ \llbracket a \rrbracket_m \in d$
(17)	$\llbracket a \rightarrow b;\ c \rrbracket == \llbracket b \rrbracket$	if $\llbracket a \rrbracket = \text{true}$	$a \in p_0$,
	$== \llbracket c \rrbracket$	if $\llbracket a \rrbracket = \text{false}$	$b,c \in d$
	$== \emptyset$		otherwise.

Several clarifications are in order at this point.

1) The distinction of name usage in (5), (14) and (15) is as follows. In (5), the evaluation of "@n" is the intensional descriptor of the name n in the current database state. In (14), the evaluation of the name n is the object described by the intensional descriptor and depends on the current database state. In (15), the evaluation of "#n" is an object described by the extensional descriptor of the name n.

2) Note that in (6), "x" may not be unique, therefore [ ] must be considered as a multivalued function. The iota operator (ι) means "some value of x such that." Similarly in (12), "a" may not be unique.

5. Semantic Consistency

The fundamental law of semantic consistency in ADS states that the extension of each name in a database must be consistent with its intension. If a name denotes a transformation, the extensional transformation must be a partial function of the intensional transformation. The symbols in the evaluation of the extension must be a subset of the symbols in the evaluation of the intension. If a name denotes a condition, the symbols satisfying the extensional condition must be a subset of the symbols satisfying the intensional condition. By definition, a database state is a consistent (legal) state if every extension of a name is consistent with the intension of the name. If a name has only an intension and therefore no extension, the name has no direct effect on the legality of the database state. The above informal discussion on database consistency is exemplified below by the predicate

Cons on M. The predicate contains two parts (A and B) which are explained below.

$$\text{Cons} : M \rightarrow 2$$

$$\text{Cons}(m) == (\forall n \in Na)(m_e(n) \neq \emptyset \quad \Rightarrow$$

$$\begin{array}{ll}
\text{Part A} & \left\{ \begin{array}{l}
\qquad (\forall k > 0) \\
\qquad\quad (\ (m_i(n) \in t_k \quad \Rightarrow \\
\qquad\qquad\quad (\forall x \in B^k)(\forall y \in B) \\
\qquad\qquad\qquad\quad (y = \llbracket m_e(n) \rrbracket_m(x) \Rightarrow \\
\qquad\qquad\qquad\qquad y = \llbracket m_i(n) \rrbracket_m(x)\ ))
\end{array} \right. \\
\text{Part B} & \left\{ \begin{array}{l}
\qquad\quad \wedge\ (m_i(n) \in p_k \quad \Rightarrow \\
\qquad\qquad\quad (\forall x \in B^k) \\
\qquad\qquad\qquad\quad (\llbracket m_e(n) \rrbracket_m(x) \Rightarrow \\
\qquad\qquad\qquad\qquad \llbracket m_i(n) \rrbracket_m(x)\ )))
\end{array} \right.
\end{array}$$

where $f(x) == f(x_1)(x_2)\ldots(x_k)$ when $x == (x_1, x_2, \ldots, x_k) \in B^k$. Note that

1) Consistency is defined only for a name with a defined extension ($m_e(n) \neq \emptyset$).

2) If the intension of n is a kth order transformer, any symbol y in the evaluated extension of n must also be in the evaluated intension of n (part A above).

3) If the intension of n is a kth order predicate, any evaluated extension of n which is true must also be true when the intensional definition of n is evaluated (part B above). Note that if the evaluated extension of n is false, we do not care what its

intensional evaluation is.

6. Update Evaluation

The initial database state $m_0 \varepsilon M$ has no intension and no extension, i.e., $(\forall n \varepsilon Na)(m_i(n) = m_e(n) = \emptyset)$. A user enters new information into the database; i.e., updates the database, by modifying the intension or the extension of a name. The update operation transforms one consistent database state into another. We denote the update evaluation function by "E", the database state before update by m, the database state after update by m' and the intensional definition of n after the update by $m'_i(n)$. Below, "<==>" means "if and only if by definition".

(1) $E : \text{update} \rightarrow (\text{Cons}^{-1}(T) \rightarrow \text{Cons}^{-1}(T))$ update evaluation

(2) $E(\vdash n{==}d)(m) = m' \iff m'_i(n){=}d$

(3) $E(\vdash n{:=}d)(m) = m' \iff m_i(n)\neq\emptyset \wedge m'_e(n){=}\text{univ}(d,m)$

where

1) $\text{Cons}^{-1}(T)$ is a consistent database; that is,

$$\text{Cons}^{-1}(T) == \{ m \varepsilon M \mid \text{Cons}(m) = T \}$$

2) $d'{==}\text{univ}(d,m)$ is a universal descriptor for d satisfying the following condition: $(\forall m' \varepsilon \text{Cons}^{-1}(T))([\![d]\!]_m = [\![d']\!]_{m'})$; that is, d' is a descriptor which contains no names.

In order for a descriptor to be universal, it must be independent of the database state m. This can not be true so long as there are names in the descriptor because the names are not evaluated until the time of usage, a time when the database state may have changed. The existence of such a universal descriptor is guaranteed by the following theorem which can be proved by induction on the number of update operations required to reach the consistent database state m from the initial state:

$$(\forall d \in D)(\forall m \in Cons^{-1}(T))(\exists d' \in D)(\forall m' \in Cons^{-1}(T))([\![d]\!]_m = [\![d']\!]_{m'}).$$

With the fixed-point operator, it is possible to write descriptors, even recursive ones, without the use of names. Appendix III of this report shows how the fixed-point operator can be expressed in ADS.

In (1) above, E is defined to be a function which evaluates to a new consistent database given an update statement and a consistent database. (2) states that the intensional definition of a name (n==d) results in a new database state m' if and only if a name had no intensional definition prior to the update but has one after the update (defined by the update statement). (3) states that the extensional definition of a name (n:=d) must result in a new database state m' if and only if the name already had an intensional definition prior to the update and the new state is consistent with the intensional definition of n, a definition whose application can depend on the database state at the time of definition.

7. Query Evaluation

An ADS database management system always responds to a user query by generating a set of symbols. Information stored in the database can be retrieved by a descriptor specifying a set of symbols. There are two ways of specifying a set of symbols through a descriptor: either by a transformer or by a predicate. When a query is made through a transformer, the range of the function will be generated. When a query is made through a predicate, the subset of the domain values that satisfy the predicate will be generated. The ordering of set element generation is unspecified. We denote the query evaluation function by the same "E" as for the update evaluation function.

$$E : \text{query} \rightarrow (M \rightarrow \bigcup_{k=0}^{\infty} B^k) \qquad \text{query evaluation function}$$

$$E(\ ?t_k\ )(m) == \{\ [t_k]_m(x)\ |\ x \in B^k\ \} \qquad k \geq 0$$

$$E(\ ?p_k\ )(m) == \{\ x \in B^k\ |\ [p_k]_m(x)\ \} \qquad k \geq 0$$

Note that a 0-th order transformer will generate a singleton set, and a higher order transformer will generate a set of symbols. A first order predicate will generate a set of symbols, and a higher order predicate will generate a set of symbol sequences. If the set specified by a query is infinite, the database management system will not terminate.

8. Conclusion

The ADS data model has a simple definition: the formal definition of a subset containing all of the fundamental concepts has been given in this report. A formal definition of the entire ADS model would entail no more conceptual difficulty; rather it would be a straightforward extension of the approach taken here (although the task might be tedious). Yet, it is a powerful model with capabilities for abstraction, consistency checking, and unstratified control.

References

[1] Church, A., Introduction to Mathematical Logic, Volume 1, Princeton University Press, 1956.

[2] Kimura, T. D., W. D. Gillett and J. R. Cox, Jr., "A Design of a Data Model Based on Abstraction of Symbols," To be published in The Computer Journal.

[3] Stoy, J., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, MIT Press, 1979.