Report Number: WUCS-81-03

1981-05-01

Formal Specifications of ADS, An Abstract Database System

Authors: Takayuki D. Kimura and Will D. Gillett

Formal Specifications of ADS, An Abstract Database System

Takayuki Kimura and Will Gillett

WUCS-81-03

May 1981

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This work was supported in part under the NCHSR Grant HSO3792 and the NIH under Grant RROO396.

Formal Specifications of
ADS, An Abstract Database System

1. Introduction

This technical report describes the formal aspects of a semantic database model, ADS, which can be used as a fundamental building block in the development and/or analysis of information systems. ADS is a formal model of information, and in the form presented here, it is not directly implementable for use within an information system.

ADS can be used in several different ways. As a formal model, it can be used as a specification tool for stating the semantics of an information system; this is useful for analysis of an information system and for comparison between different information systems. With some modifications (to restrict its power), ADS can be implemented and used as the kernel of an information system. If such a kernel is developed, a significant interface system must also be developed to "buffer" the end-user from the rather terse notation used by ADS. In any event, ADS is not intended to be used directly by any end-user. In the remainder of this report, the term "user" refers to any process that communicates directly with ADS and does not designate the end-user.

Section 2 presents some fundamental concepts about our notion of a database system. Section 3 presents an overview of ADS. Sections 4, 5, 6 and 7 present formal definitions, a meta-language for ADS, the syntax of ADS and the semantics of ADS, respectively. Section 8 presents some examples.

2. Fundamental Notions

A database is a collection of symbols denoting a collection of judgements (in the broadest sense of the word) made by a community of users about their knowledge of a portion of the world in which they share an interest. Our use of the word judgement includes the report of simple measurements as well as a decision requiring wisdom or skill. Because of the symbolic representation of these judgements, a database is a linguistic entity with which there exist associated rules of syntax and semantics. A database model is a model of the linguistic structure of the symbolism by which judgements are represented.

Note that a database is not a model of the world; rather, it is a model of the user's view (knowledge) of the world. Recognition of the difference is essential in constructing a conceptual model of a database. What matters is not the correspondence between the database and the real world, but between the database and the user's view of the world. While reality stays the same, the view of reality may change, and vice-versa. A database model must be able to reflect such a change instead of forcing a particular view of the world onto the users. When there exists more than one view of the same reality within the same user community, it is a database system that preserves the consistency among the different views representable in the database.

A database system, of which a database is a part, is a communication mechanism through which the users can share multiple views of the world. Two users communicating through a database system may be the same process at two different time instances, or they may be two distinct processes which might be geographically distant from one another. The utility of a database system stems, first, from its capability to check and enforce the agreements among the users on judgements about the world, and secondly, from its capability of conveying

the records of judgements from one point in time/space to another.

Since all judgements are represented in the database by symbols, the agreements on judgements can be reduced to the agreements on symbol usage; i.e., a grammatical rule and an interpretation rule of the symbolism. A way of viewing the world is defined by symbol usage. A fundamental function of a database system is to check the consistency, both syntactic and semantic, of a newly entered declaration (symbolic representation of a judgement) with the declarations that already exist in the database. In order to perform this function, a database system must contain the syntactic and semantic rules of symbolism assumed by the users.

Furthermore, the system should be able to change these rules under the user's direction, reflecting the changes in the user's view of the world.

There are two kinds of judgements that a user can communicate to the data base system: intensional judgements and extensional ones. From a logical point of view, an intensional judgement involves general terms and 'concepts', and an extensional one involves specific terms and 'individuals'. The former represents the user's view of how the world is supposed to be, and the latter represents the user's view of how the world actually is. In linguistic terms, an intensional judgement entails a user's commitment to particular rules of symbol usage (possibly by updating the existing rules), and an extensional one implies an actual usage of symbols. Extensional judgements are required to be consistent with intensional judgements in the same way that actual use of symbols must be consistent with the prior commitment to a particular grammar. A database system checks this consistency and rejects inconsistent judgements. In this regard, a database system can be considered as a language processor (eg., a compiler)

with a significant amount of intelligence.

It should be noted that whether a particular judgement is extensional or not depends upon the level of abstraction from which the decision is to be made. The situation is similar to the one encountered in deciding whether a particular symbol belongs to an object language, or a meta-language, or a meta-meta-language, and so on. For example, if a user has decided how syntactic and semantic rules of symbolism should be stated, and has entered this judgement to the database system as declaration A (an intensional one), then declaration B, which makes a particular change in grammar, can be considered as an extensional one with respect to A; however, B can be considered as an intensional one with respect to a judgement involving a symbol usage. It is also important to note that the above observation is relevant only when the database is capable of accepting and representing judgements about symbolism in general, and about the symbolic representation of other judgements within the database in particular, i.e., a self-referencing capability.

3. Abstract Database System Overview

The Abstract Database System (ADS) is a mathematical model of a database. The model presented here has the self-referencing capability and is a single-user model. The model serves several purposes: (1) It is a stepping stone to a more general model for a multi-user community where different but important issues, such as concurrency control, access control and 'external views' of different users, must be addressed. (2) The validity of the observations made in the previous section about modeling a database can be tested. (3) While ADS is a mathematical model, it can indicate the nature of the resources required for implementing a database system whose design is based on a similar model. (4) Since ADS

is intended to be 'elementary' in the sense of Kent [4], it can serve as a reference point in comparing the power and limitations of known 'secondary' or 'vernacular' models such as those proposed by Codd [3], Chen [2], Senko [6], Smith and Smith [7], Abrial [1], and Schmid [5]. (The examples given at the end of this appendix demonstrate this point.)

The model is called 'abstract' because it is a mathematical model, and has no physical structure associated with it. This is not to say, however, that the implementability or feasibility of the final model has been neglected. On the contrary; we are well aware of which aspects of ADS have straightforward implementations and which do not. Our selection of features to include in ADS has been significantly affected by implementation constraints.

In the remainder of this section we will give an informal overview of the ADS, and in the following three sections, a particular ADS based on the set of binary trees will be described mathematically.

In its most general form, an ADS is a state machine (automaton) depicted as in Figure 1.

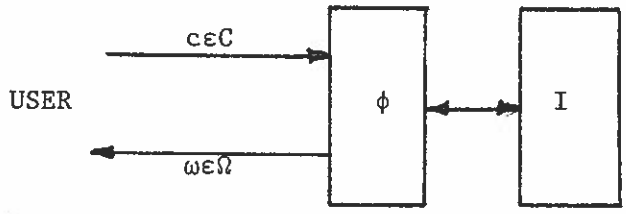


Figure 1: ADS as a state machine.

where: I (information state space): the (infinite) set of information states,

C (commands): the set of input symbols,

Ω (objects): the set of output symbols,

ϕ (interpretation): the next state function, i.e., $\phi: C \times I \rightarrow \Omega \times I$.

The user enters a sequence of judgements and queries into the system by

choosing a sequence of commands. Each command is interpreted by ϕ, and if the command is 'consistent' (to be defined later) with the current information state, then the command is accepted (otherwise rejected), and the user is so notified. If the command represents a user judgement, either intensional or extensional, ϕ will record it by changing the information state. If the command represents a user query, ϕ will construct a response and display it to the user without modifying the information state. Thus, the information state in the ADS corresponds to the conventional concept of 'database'. It should be pointed out here that the information state is encoded by symbols denoting intensional judgements as well as extensional ones; i.e., the information state can be thought of as containing the syntactic and semantic rules by which Φ interprets the symbols contained in the information state.

The next level of the ADS description specifies the structure of I, C, Ω, and ϕ.

For each ADS, there exists a base language B in which the information structure (or the content of the database) is represented (or encoded). The language B is an object language in the sense that it has its own syntax but no semantics by itself. It is simply a set of patterns, and any algebraic system can be a base language as long as the set satisfies a certain set of axioms (to be given later). In the particular ADS to be described in the following sections, we have chosen the set of all binary trees as the base language, mainly because it is the simplest algebraic system satisfying the required property and also it is a well-known set.

An information state, a member of I, is represented by a finite set of expressions from B. The interpretation of those expressions is carried out by ϕ based on the semantic rules represented in the information state itself.

Since the information state is modified by user transactions, the semantics of an information state is under the user's control and changes dynamically. The structure of ϕ, on the other hand, depends upon the structure of B, but it is independent of user transactions.

The user expresses his/her judgements and queries in the first order language C, which is syntactically a sublanguage of B with the semantics defined by both ϕ and the current information state. The syntax of C is fixed for each ADS. The views of the world that the user can communicate to the system are primarily dictated by the syntax of C.

As far as the ADS is concerned, a view of the world is significant, when and only when it is representable in terms of the elements of B, the subsets of B, and the relationships among them. That is to say, the universe of discourse for an ADS, denoted by Ω and called the set of objects, consists of elements (of B), sets (of elements) and assertions (logical values).

The semantics of C and the information state are defined on the domain of these objects.

The domain of ϕ is actually a superset of C and is called the language L. The alphabet of L is called the symbols Σ and is partitioned into logical symbols Λ, variables V, and names N.

Under the interpretation of ϕ, logical symbols, such as logical connectives, quantifiers and the equality sign, have the usual fixed meanings. The range of variables is over B, i.e., over the elements. A name denotes an object, when so defined by the user.

The language L contains another sublanguage called descriptors D. A descriptor also denotes an object. The association between a descriptor and

its denotation (object) is inherent in the structure of the descriptor, while the association between a name and its denotation is purely incidental and only the user can establish the association.

An information state is a collection of user defined names of objects, called the proper names P, along with the two attributes, the intension μ, and the extension τ.

The intension of a proper name is specified by a descriptor which is given by the user when the name is defined. The extension of a proper name can be defined only after the name and its intension have been defined. The extension must always be consistent with the intension in the following sense: let n be a proper name, $\mu(n)$ be its descriptor, $\tau(n)$ be its extension, and $\phi(\mu(n))$ be the object described by the descriptor $\mu(n)$. If n is an element name, then it must be that if $\tau(n)$ is defined then $\tau(n) = \phi(\mu(n))$. If n is a set name, then it must be that $\tau(n) \subseteq \phi(\mu(n))$. If n is an assertion name, then it must be that $\tau(n) = \phi(\mu(n))$.

Generally speaking, the intension of a name specifies what the name can possibly mean. The extension specifies what the name currently means.

A user command to ADS can be either an update command or a query command. An update command changes the information state by either adding a new judgement or cancelling an old one. An update command can be also either intensional, i.e., defining a new name with its descriptor, or extensional, i.e., assigning an extension to an already defined name. A query command is an expression in L which denotes an object. The system displays the specified object.

A transaction is an indivisable sequence of update commands. The user can enter any sequence of update commands within a transaction and change the information state as long as the set of assertions in the information state whose extensions are defined satisfy the following invariant condition at the end of the

transaction.

Let a be an arbitrary assertion name. If $\tau(a)$ is defined (i.e., the user had assigned the extension), then $\phi(\mu(a)) = \tau(a)$, i.e. ϕ evaluates the descriptor $\mu(a)$ to the same logical value as $\tau(a)$.

Thus, the set of assertions whose extensions are defined by the user plays the same role as the 'constraint set' in the conventional database model.

The initial information state is an important part of the ADS definition. It must contain an appropriate set of objects (and proper names) in order that the user may exploit fully the system capabilities, in particular, the self-referencing capability.

4. Formal Definitions

In this section we will define an ADS utilizing the set of binary trees as the base language. A meta-language for L is given in the next section.

(i) Binary Trees.

Let B be the set of all binary trees defined by the following production rules on the alphabet {0,1}:

(a) $0 \in B$,

(b) if $\alpha, \beta \in B$, then $1\alpha\beta \in B$,

(c) nothing else is in B.

Note: B = {0, 100, 11000, 10100, 1100100, ...} represents {·,  ,  , ,  , ...}

Let T_X be the set of all binary trees with the leaf nodes in X, where X is an arbitrary subset of B, i.e., the production rules for T_X are as follows:

(a) if $\alpha \in X$, then $\alpha \in T_X$

(b) if $\alpha, \beta \in T_X$ then $1\alpha\beta \in T_X$

(c) nothing else is in T_X.

Notes: (1) $T_X \subseteq B$ for any $X \subseteq B$.

(2) $B = T_{\{0\}}$.

(3) B can be considered as an algebraic system <B, +, {0}>

where +: B×B → B is a binary operator, and {0} is a generator of B satisfying the following axioms:

(a) $(\forall u,v,x,y \in B)(+(u,v) = +(x,y) \Rightarrow (u = x \wedge v = y))$

(b) $(\forall x,y \in B)(+(x,y) \notin \{0\})$

(c) $(\forall x \in B-\{0\})(\exists y,z \in B)(x = +(y,z))$

Furthermore let

$2 \triangleq \{True, False\}$: the set of logical values

$2^B \triangleq \{X | X \subseteq B\}$: the set of all subsets of B

$\Omega \triangleq 2 \cup B \cup 2^B$: the set of objects

Now, we are ready to define an ADS.

(ii) Abstract Database System based on B: ADS_B.

ADS_B is an 8-tuple <B, Σ, L, D, C, I, ϕ, I_o>, where

$B = T_{\{0\}}$: base language (the set of binary trees)

$\Sigma \subseteq B$: symbols (a decidable and uniquely deconcatenable* subset of B).

$\Sigma = \Lambda \cup V \cup N$ (disjoint union of Λ, V, and N)

where Λ: logical symbols,

V: variables,

N: names (constants),

$L \subseteq T_\Sigma$: Language (a collection of structured sets of symbols)

*no member of Σ is a subtree of another member of Σ.

$D \subseteq L$: descriptors (a descriptive sublanguage of L for denoting objects)

$D = D_A \cup D_E \cup D_S$

where D_A: assertion descriptors,

D_E: element descriptors,

D_S: set descriptors.

$C \subseteq L$: commands (a prescriptive sublanguage of L for denoting user commands)

$C = C_U \cup C_Q$

where C_U: update commands,

C_Q: query commands.

$I \subseteq \{<P, \mu, \tau> \mid P \subseteq N$: proper names;

μ: $P \to D$: intension of proper names;

τ: $P \to \Omega$ (partial): extension$\}$:

information states

where if $<P, \mu, \tau> \in I$,

then $\tau(a) \subseteq 2$, for $a \in A$

$\tau(e) \subseteq B$, for $e \in E$

$\tau(s) \subseteq 2^B$, for $s \in S$

where $A \triangleq \mu^{-1}(D_A)$: assertion names,

$E \triangleq \mu^{-1}(D_E)$: element names,

$S \triangleq \mu^{-1}(D_S)$: set names.

ϕ: $L \times I \to \Omega \times (I \cup \Omega)$ (partial): interpretation of L

where ϕ: $D_A \times I \to 2$

$D_E \times I \to B$

$D_S \times I \to 2^B$

$C_U \times I \to 2 \times I$

$C_Q \times I \to \Omega$

$I_o \in I$: initial information state

Note: Since B, Σ, Λ, V, N, L, D, C, P, μ, τ, A, E, S, φ are all subsets of B (with proper encoding for μ, τ, φ), these sets can be made available to the user by including their names and the proper descriptors in the initial information state.

5. A Meta-language for L

The language L is defined as a subset of B, and it is awkward to talk about its syntax and semantics directly. Therefore, we introduce a meta-language for L, denoted by *L*, whose mapping to binary trees is straightforward.

The syntax of *L* will be given first, then the evaluation function of *L*, representing the semantics of L and the structure of φ, will be given afterwards.

5.1 Syntax (BNF) of *L*

Logical symbols (Λ): (,),. A(∀), E(∃), L(λ),
I(ι), #(ε), ⊢, ~⊢ , ?, +, ;, ', ",
=, ==, =>, <-, TAU(τ), MU(μ),
PHI(φ)

Variables (V): any alphanumeric word

Names (N): any alphanumeric word (in the BNF, P is classified into A_NAMES, E_NAMES, and S_NAMES)

The following is an LALR(1) grammar for the meta-language *L*.

1) COMMANDS

```
session ::=      command_list

command_list ::=command |
                 command_list ; command

command ::=      transaction |
                 query

transaction ::= < update_list >

update_list ::= update
                update_list ; update |

update ::=       |- definition |
                 |- fact |

                ~|- defined _ name |
                ~|- fact
```

2) QUERIES

```
query ::=        ? expression
```

3) DEFINITIONS and FACTS

```
definition ::=  name == e_expression

defined_name ::=name

fact ::=         A_NAME <- a_expression |
                 E_NAME <- e_expression |
                 S_NAME <- ( s_expression ) |
                 S_NAME <- e_expression
```

4) EXPRESSIONS

```
expression ::=  a_expr |
                e_expr |
                s_expr |
                f_expr

a_expression ::=a_express |
                a_designator

e_expression ::=e_express |
                e_designator

s_expression ::=s_express |
                s_designator
```

```
a_express ::=   a_descriptor |
                e_expression = e_expression |
                e_expression # s_expression |
                ( a_expression => a_expression ; a_expression )

e_express ::=   VARIABLE |
                e_descriptor |
                " lexical_unit " |
                ' lexical_unit ' |
                + e_expression e_expression

s_express ::=   s_descriptor

f_expr ::=      TAU ( e_expression ) |
                PHI ( e_expression )

a_expr ::=      a_express |
                a_desig

e_expr ::=      e_express |
                e_desig

s_expr ::=      s_express |
                s_desig
```

5) DESIGNATORS

```
a_designator ::=a_desig |
                f_expr

e_designator ::=e_desig |
                f_expr

s_designator ::=s_desig |
                f_expr

a_desig ::=     A_NAME |
                T |
                F

e_desig ::=     E_NAME |
                NIL |
                MU ( e_expression )

s_desig ::=     S_NAME |
                B |
                NULL
```

6) DESCRIPTORS

```
a_descriptor ::=( A form ) ( a_expression ) |
               ( E form ) ( a_expression )

e_descriptor ::=( I form ) ( a_expression )

s_descriptor ::=( L form ) ( a_expression )
```

7) FORMS

```
form ::=        form1 # type |
                form2

form1 ::=       VARIABLE |
                + form1 form1

form2 ::=       + form3 form3 |
                + form3 form2 |
                + form2 form3

form3 ::=       ( VARIABLE # type )
```

8) MISC.

```
type ::=        s_expression

lexical_unit ::=form2 |
                expression

name ::=        A_NAME |
                E_NAME |
                S_NAME
```

5.2 Semantics for L (Specification of ϕ)

Let IS=<P,μ,τ> ε I be an arbitrary information state. Let [δ] be the evaluation of δ ε L under IS; i.e., $[\delta] \underset{\Delta}{=} \phi(\delta, IS)$ where δ ε L and IS ε I.

The evaluation function [ ] is defined recursively for each component of L; we adopt the following notations:

n, n_e, n_s, n_a: an arbitrary name, e_name, s_name, or a_name.

d, d_e, d_s, d_a: an arbitrary descriptor, e_descriptor, s_descriptor, or a_descriptor.

e_expr,s_expr,a_expr: an arbitrary e_expression, s_expression, or a_expression.

$\gamma,\gamma_1,\gamma_2,\ldots,\gamma_n$: arbitrary expressions.

$x,x_1,x_2,\ldots,x_n$: arbitrary variables.

$t,t_1,t_2,\ldots,t_n$: arbitrary types (sets).

$\alpha(x_1,x_2,\ldots,x_n)$: an arbitrary form with $n \geq 1$ typed variables ($x_i \in t_i$ is assumed).

$\beta(x_1,x_2,\ldots,x_n)$: an arbitrary a_expression containing $n \geq 1$ free variable.

$\beta(\gamma_1,\gamma_2,\ldots,\gamma_n)$: the result of free substitution of γ_k for x_k in $\beta(x_1,x_2,\ldots,x_n)$, $1 \leq k \leq n$. Free substitution is allowed for any variable not contained in a double quote ("). Free substitution places a quote around the substituted expression.

1) COMMANDS

[command_list] $\overset{\Delta}{=}$ [command$_1$] then
[command$_2$] then
.
.
.
[command$_n$]

where command$_i$ is in command_list in the sequence indicated (no response to the user).

[transaction] $\overset{\Delta}{=}$ [<] [update_list] [>]

[<] $\overset{\Delta}{=}$ save a copy of the current information state in saved_IS (no response to the user).

[>] $\overset{\Delta}{=}$ Let RESP $\equiv \Lambda, \phi(\mu(\text{assert})) = \tau(\text{assert})$
{assert $\in$ A_NAMES |
τ(assert is defined}
if RESP then respond (accept)
else respond (reject)
replace IS by saved_IS

[update_list] = [update$_1$] then
[update$_2$] then
.
.
.
[update$_n$]

where update$_i$ is in update_list in the sequence indicated (no response to the user).

2) UPDATE COMMANDS

```
[⊢ n==e_expr]   ≜Δ if n ε P then respond (reject)
                   else if [e_expr] ε E_DESC then
                        E_NAMES´ = E_NAMES ∪{n}
                        μ´(n) = [e_expr]
                        τ´(n) = UNDEF
                        respond (accept)
                   else if [e_expr] ε A_DESC then
                        A_NAMES´ = A_NAMES ∪{n}
                        μ´(n) = [e_expr]
                        τ´(n) = UNDEF
                        respond (accept)
                   else if [e_expr] ε S_DESC then
                        S_NAMES´ = S_NAMES ∪{n}
                        μ´(n) = [e_expr]
                        τ´(n) = UNDEF
                        respond (accept)
                   else respond (reject)

[~⊢ n]          ≜Δ if n ε P then
                        μ´(n) = UNDEF
                        τ´(n) = UNDEF
                        P´ = P-{n}
                        respond (accept)
                   else respond (reject)

[⊢ n_e<-e_expr] ≜Δ if n_e ε E_NAMES and
                        τ(n_e) = UNDEF and
                        [e_expr # (Lα)(β)] = True
                        (where α and β are such that
                        μ(n_e) = (ια)(β))
                   then
                        τ´(n_e) = [e_expr]
                        respond (accept)
                   else respond (reject)

[~⊢ n_e<-e_expr] ≜Δ if n_e ε E_NAMES and
                        τ(n_e) = [e_expr]
                   then
                        τ´(n_e) = UNDEF
                        respond (accept)
                   else respond (reject)

[⊢ n_a<-a_expr] ≜Δ if n_a ε A_NAMES and
                        τ(n_a) = UNDEF and
                        [μ(n_a)] = [a_expr]
                   then
                        τ´(n_a) = [a_expr]
                        respond (accept)
                   else respond (reject)
```

$[\sim\vdash n_a \leftarrow a\_expr] \triangleq$ if $n_a \in$ A_NAMES and $\tau(n_a) = [a\_expr]$
then
$\tau'(n_a)$ = UNDEF
respond (accept)
else respond (reject)

$[\vdash n_s \leftarrow e\_expr] \triangleq$ if $n_s \in$ S_NAMES and $[e\_expr] \in [\mu(n_s)]$
then
$\tau'(n_s) = \tau(n_s) \cup \{[e\_expr]\}$
respond (accept)
else respond (reject)

$[\sim\vdash n_s \leftarrow e\_expr] \triangleq$ if $n_s \in$ S_NAMES and $[e\_expr] \in \tau(n_s)$
then
$\tau'(n_s) = \tau(n_s) - \{[e\_expr]\}$
respond (accept)
else respond (reject)

$[\vdash n_s \leftarrow (s\_expr)] \triangleq$ if $n_s \in$ S_NAMES and
[s_expr] exists and
$[s\_expr] \cap [\mu(n_s)] \neq \Phi$ (the empty set)
then
$\tau'(n_s) = \tau(n_s) \cup ([s\_expr] \cap [\mu(n_s)])$
respond (accept)
else respond (reject)

$[\sim\vdash n_s \leftarrow (s\_expr)] \triangleq$ if $n_s \in$ S_NAMES and
[s_expr] exists and
$[s\_expr] \cap \tau(n_s) \neq \Phi$
then
$\tau'(n_s) = \tau(n_s) - [s\_expr]$
respond (accept)
else respond (reject)

3) QUERY COMMANDS

[?expression] $\triangleq$ if [expression] is defined then
respond ([expression])
else respond (reject)

4) DESCRIPTORS

Let $\alpha(z_1,z_2,\ldots,z_n)$ be an arbitrary 'form' where either z_i = "x_i" (i.e., α is a form1) or z_i = "$x_i \in type_i$" (i.e., α is a form2). Let $\eta_1,\eta_2,\ldots,\eta_n$ be arbitrary e_expressions. Define an intrinsic function RANGE as follows:

```
RANGE(α(z1,z2,...,zn)) ≡ if α is a form "form1 ε type" then
                           return ({[α(η1,η2,...,ηn)]|
                           ∃ η1,η2,...,ηn ∋ [α(η1,η2,...,ηn)] ε [type]})
                         else /* must be form2 */
                           return ({[α(η1,η2,...,ηn)]|
                           ∃ η1,η2,...,ηn ∋ [ηi] ε [typei]})
```

Intuitively RANGE(α) represents the subset of B that can be 'bound' to the form $\alpha(z_1,z_2,\ldots,z_n)$.

```
[(A α(z1,z2,...,zn))(β(x1,x2,...,xn))] ≡Δ
                         if ∀[α(η1,η2,...,ηn)] ε RANGE(α(z1,z2,...,zn))
                              [β(η1,η2,...,ηn)] = True
                         then respond (True)
                         else respond (False)

[(E α(z1,z2,...,zn))(β(x1,x2,...xn))] ≡Δ
                         if ∃ η1,η2,...,ηn ∋ [α(η1,η2,...,ηn)] ε RANGE(α(z1,z2,...,zn))
                                  and  [β(η1,η2,...,ηn)] ≡ True
                         then respond (True)
                         else respond (False)

[(L α(z1,z2,...,zn))(β(x1,x2,...,xn))] ≡Δ
                         respond ({[α(η1,η2,...,ηn)] ε RANGE(α(z1,z2,...zn))|
                                  [β(η1,η2,...,ηn)] = True})

[(I α(z1,z2,...,zn))(β(x1,x2,...,xn))] ≡Δ
                         if ∃ η1,η2,...,ηn ∋ [α(η1,η2,...,ηn)] ε RANGE(α(z1,z2,...,zn))
                                  and [β(η1,η2,...,ηn)] = True
                         then respond ([α(η1,η2,...,ηn)])
                         else respond (UNDEF)
```

5) EXPRESSIONS

```
[TAU(e_expr)]  ≡Δ if [e_expr] ε P and
                        τ([e_expr]) ≠ UNDEF
                 then respond (τ([e_expr]))
                 else respond (UNDEF)

[PHI(e_expr)]  ≡Δ respond ([[e_expr]])
```

6) A_EXPRESSIONS

```
[n_a]                 ≡Δ if n_a ε A_NAMES then
                           if τ(n_a) = UNDEF then
                                respond ([μ(n_a)])
                           else respond (τ(n_a))
                      else respond (UNDEF)

[T]                   ≡Δ respond (True)

[F]                   ≡Δ respond (False)

[e_expr1=e_expr2]     ≡Δ if [e_expr1] = UNDEF or
                            [e_expr2] = UNDEF
                      then respond (UNDEF)
                      else respond ([e_expr1] = [e_expr2])

[e_expr # s_expr]     ≡Δ if [s_expr] exists then
                                respond ([e_expr] ε [s_expr])
                      else respond (UNDEF)

[a_expr1=>a_expr2;a_expr3] ≡Δ
                      if [a_expr1] ≠ UNDEF then
                           if [a_expr1] = True then respond ([a_expr2])
                           else respond ([a_expr3])
                      else respond (UNDEF)
```

7) E_EXPRESSIONS

```
[variable]            ≡Δ respond (UNDEF)

["lexical_unit"]      ≡Δ respond (lexical_unit)

['lexical_unit']      ≡Δ respond (lexical_unit)

[+ e_expr1 e_expr2]   ≡Δ respond (1[e_expr1][e_expr2])

[n_e]                 ≡Δ if n_e ε E_NAMES then
                           if τ(n_e) = UNDEF then respond ([μ(n_e)])
                           else respond (τ(n_e))
                      else respond (UNDEF)

[NIL]                 ≡Δ respond (0)

[MU(e_expr)]          ≡Δ if [e_expr] ε P then
                                respond (μ([e_expr]))
                      else respond (UNDEF)
```

8) S_EXPRESSIONS

$[n_s]$ $\stackrel{=}{\Delta}$ if $n_s \in$ S_NAMES then
if $\tau(n_s)$ = UNDEF then
respond ($[\mu(n_s)]$)
else respond (UNDEF)

[B] $\stackrel{=}{\Delta}$ respond (the set of all binary trees)

[NULL] $\stackrel{=}{\Delta}$ respond (the empty set)

6. Examples

In this section, examples are presented to show the flavor and utility of the ADS. For ease of understanding, the examples are kept simple, and certain aspects of the model are not present (e.g. the concept of deleting judgements is not represented).

Example 1 is a fairly "dry" presentation of how the Lisp functions of car and cdr might be implemented. It is included to show how arbitrary structures are manipulated.

Example 2 deals with a simple database involving cars, their colors, and their manufacturers. It presents this database in the form of a relational or record oriented model (probably most familiar and easy to understand).

Throughout this section, we assume that the initial information state I_o contains the following:

'E_NAMES', 'A_NAMES', 'S_NAMES' ε τ(S_NAMES)

τ(E_NAMES), τ(A_NAMES) = { }

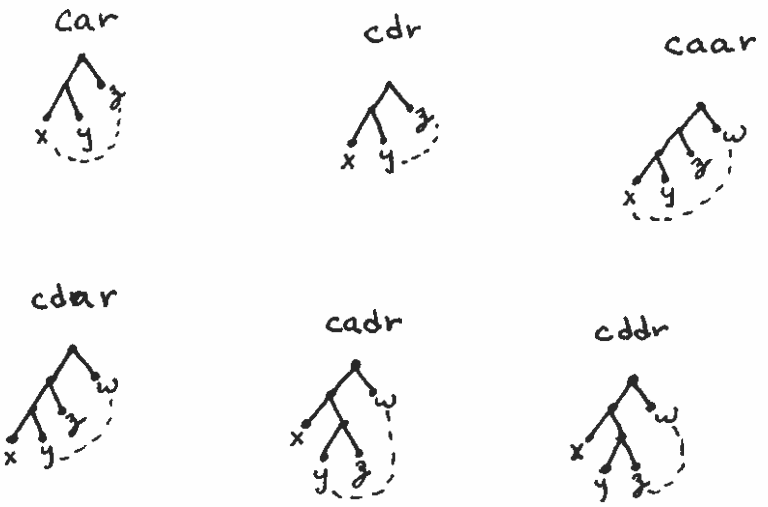
Example 1: Some Lisp Functions

```
(1)  <
(2)     ├ car=="(L ++xyz # B)(x = z)";                    → accept
(3)     ├ cdr=="(L ++xyz # B)(y = z)";                    → accept
(4)     ├ caar=="(L+++xyzw # B)(x = w)";                  → accept
(5)     ├ cdar=="(L +++xyzw # B)(y = w)";                 → accept
(6)     ├ cadr=="(L ++x+yzw # B)(y = w)";                 → accept
(7)     ├ cddr=="(L ++x+yzw # B)(z = w)";                 → accept
(8)     ├ a=="(I x # B)(T)";                              → accept
(9)     ├ a<- ++NIL +NIL NIL +NIL NIL;                    → accept
(10)    ├ b=="(I x # B)(+ax # car)";                      → accept
(11)    ├ c=="(I x # B)(+bx # cdr)";                      → accept
(12)    ├ d=="(I x # B)(+ax # cdar)";                     → accept
(13)    ├ e==MU("a");                                     → accept
(14)    ├ e<- +NIL NIL                                    → accept
(15)  >;                                                  → accept
(16)    ? TAU("b");                                       → UNDEF
(17)    ? b;                                              → +NIL +NIL NIL
(18)    ? c;                                              → +NIL NIL
```

```
(19)    ? d;
                                                  → +NIL NIL
(20)    ? (L x # E_NAMES)(PHI(x)=e);
                                                  → {c,d,e}
(21)    ? (L x # E_NAMES)( PHI(x)=e =>
              (x='e'=>F;T);F);
                                                  → {c,d}
(22)    ? (L x # E_NAMES)((I u # B)
              (+ PHI(x) u # cdar) =
              (I v# B)(+ PHI(x) v # cdr))
                                                  → {a}
```

Discussions of Example 1

Lines 2-7 define several of the selection functions available in Lisp. The patterns which they define are:



As an example, consider cdr. The right subtree of cdr (z) selects the right subtree (y) of the left subtree ( ).

Line 8 defines an element name, a, whose intension (and therefore also its extension) can be any binary tree. Line 9 associates the name 'a' with the specific binary tree

Lines 10-12 logically define functions (with input parameters b, c and d, respectively) which use the previously defined patterns to select the car, cdr and cdar from the input parameter. Note that only their intensions are defined.

Line 13 defines a new element name, e, with the identical intension of that of a. Line 14 associates a different member of B with e.

Since the extension of b have not been specified, line 16 indicates that it is undefined. However, the intension of b, c and d are well defined and are shown in lines 17-19.

Line 20 asks "What element names have the same 'interpretation' as that of e?" The answer, of course, is c, d and e. Line 21 asks "What element names other than e have the same 'interpretation' as that of e?" Line 22 asks "What element names have an 'interpretation' such that its cdr is the same as its cdar?"

Example 2: Car-color-maker database (a la Codd)

```
car = record
      u: name
      v: color
      w: maker
    end
```

u	v	w	attribute
name	color	maker	domain
C1	blue	Chev	
C2	red	Chev	
C3	blue	Chev	
C4	red	Ford	

```
(1)  <
(2)      ├ name=="(L x # E_NAMES)(T)";                                    → accept
(3)      ├ color== MU("name"); ├ maker== MU("name");                      → accept^2
(4)      ├ C1=="(I x # B)(T)";                                            → accept
(5)      ├ C2==MU("C1"); ├ C3==MU("C1"); ├ C4==MU("C1");                  → accept^3
(6)      ├ name<-"C1"; ├ name<-"C2"; ├ name<-"C3"; ├ name<-"C4";          → accept^4
(7)      ├ blue==MU("C1"); ├ red==MU("C1"); ├ yellow==MU("C1");           → accept^3
(8)      ├ color<-"red"; ├ color<-"blue"; ├ color<-"yellow";              → accept^3
```

```
(9)     ├ Chev==MU("C1"); ├ Ford==MU("C1");
(10)    ├ maker<-"Chev"; ├ maker<-"Ford";
(11)    ├ car=="(L +(u # TAU("name"))
                   +(v # TAU("color"))
                    (w # TAU("maker")))
              (T)"
(12)  >;
(13)    ? car;
(14)    ? TAU("car");
(15)  <
(16)    ├ car<- + "C1" + "blue" "Chev";
(17)    ├ car<- + "C2" + "red" "Chev";
(18)    ├ car<- + "C3" + "blue" "Chev";
(19)    ├ car<- + "C4" + "red" "Ford";
(20)    ├ car<- + "C5" + "blue" "Ford"
(21)  >;
(22)    ? TAU("car");
(23)  <
(24)    ├ mycar=="(I x # TAU("car"))
                  (x # (L +u+vw # B)(u = "C2"))";
(25)    ├ yourcar=="(I x # TAU("car"))
                    x # (L +u+vw # B)
                       ((v="red"=>w="Ford";F)))"
(26)  >;
(27)    ? mycar;
(28)    ? yourcar;
(29)  <
```

```
→ accept²
→ accept²
→ accept
→ accept
→ {a set containing
   4×3×2 binary trees}
→ UNDEF
→ accept
→ accept
→ accept
→ accept
→ reject
→ accept
→ {+C1+blue Chev,
   +C2+red Chev,
   +C3+blue Chev,
   +C4+red Ford}
→ accept
→ accept
→ accept
→ +C2+red Chev
→ +C4+red Ford
```

```
(30)    ├ name_of=="(L ++u+vwz # B)(u = z)";                    → accept
(31)    ├ color_of=="(L ++u+vwz # B)(v = z)";                   → accept
(32)    ├ make_of=="(L ++u+vwz # B)(w = z)"                     → accept
(33)  >;                                                        → accept
(34)     ?+mycar "blue" # color_of;                             → False
(35)     ?+mycar "red" # color_of;                              → True
(36)     ?(I x # maker)(+yourcar x # maker_of);                 → Ford
(37)     ?(L +(x # color)(y # name))
          (∃ z # maker)((+mycar Z # maker_of=>
                        +y+xz # TAU("car");F));
                                                               → {+ blue C1,
                                                                  + red C2,
                                                                  + blue C3}
(38)     ?(L x # TAU("color"))(A y # TAU("car"))
                              ((+yx # color_of=>F;T))
                                                               → {yellow}
```

Discussion of Example 2

Lines 2-3 define sets whose extensions will contain the names, colors and makers of the cars in the database. Lines 4-5 define element names; these are placed in τ(name) in line 6. Lines 7-10 complete the definitions of color and maker in a similar manner.

Line 11 defines the intension of the set name car to be 3-tuples whose components are selected from τ(name), τ(color) and τ(maker), in that order; there is no restriction on the relationship between components in the tuples (such as "Ford does not make yellow cars"). The intension of this set corresponds to the cross product of the three sets τ(name), τ(color) and τ(maker), as indicated by line 13; however, τ(car) is currently undefined, as indicated by line 14.

Lines 16-19 define specific cars. Line 20 is rejected because C5 is not a member of τ(name), and thus the triple cannot be 'bound' to the form of car.

Line 24 defines mycar to be that element of τ(car) such that its name is C2. Line 25 defines yourcar to be that element of τ(car) that is a red Ford.

Lines 30-32 define patterns for selecting the name of, color of, and maker of specific cars. Line 34 asks "Is mycar blue?"; the answer is "No". Line 35 asks "Is mycar red?"; the answer is "Yes". Line 36 asks "Who is the maker of yourcar?"; the answer is Ford. Line 37 asks "What are the colors and names of the cars made by the maker of mycar?" Line 38 asks "What colors are associated with none of the cars?"