CAELAB Memo 94-01 March 1994

AMOS.v1 User's Guide

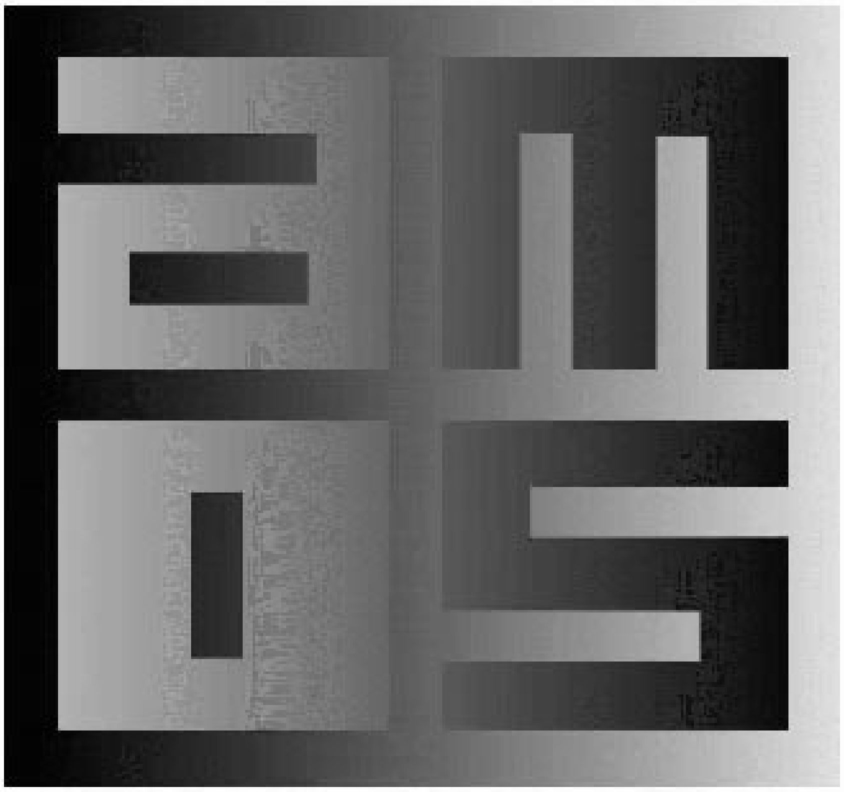
Jonas S. Karlsson, Staffan Larsson, Kjell Orsborn, Tore Risch, Martin Sköld, Magnus Werner

Dept. of Computer Science, Linköping University, 581 83Linköping, Sweden

Abstract

AMOS (Active Mediating Object System) is an Object-Relational database system. AMOS differs from the first generation Object-Oriented (OO) databases in that a relationally complete query language, AMOSQL, is available which is more general than relational query languages, such as SQL. Furthermore, AMOS is a main-memory database system, since the design of AMOS is optimized for efficient execution assuming that the entire database fits in main memory. For persistence, the system provides primitives for logging and for saving and restoring the database from disk. AMOS is implemented in C and runs on HP and SUN Unix platforms.

This manual describes how to use the AMOSQL query language. For interfaces to C, Lisp, and description of some internals, see **AMOS System Manual**.



1 Introduction

AMOS (Active Mediating Object System) is an Object-Relational database system with roots in the IRIS data model and its query language, OSQL[1] . The AMOS implementation is an extension and modification of the WS-IRIS system[1]. AMOS is optimized for databases where the entire database fits in main memory [2]. The architecture gives AMOS excellent performance (as shown for AMOS' predecessor, WS-IRIS, in [1]) while retaining a relationally complete declarative query interface to the database. For persistence, the system provides primitives to save and restore the database from disk. A transactional logging system is optionally available which logs committed database transactions on disk and thus makes all transactions recoverable after main-memory crashes.

The system can either be run in single user mode, or as a multi-user single-threaded server with restricted concurrency. The system includes client communication primitives whereby a client process can communicate with AMOS over a network. The client interface is documented in[7].

A graphical interface for AMOS also exists. However, this is only a prototype and will be redesigned and reimplemented in the future.

AMOS' query language, called AMOSQL, is an extension of the OO declarative query language of IRIS, termed OSQL. The main new features of AMOSQL are generalized foreign functions, a limited form of recursion, resolution of late binding of overloaded functions[9], an efficient execution strategy for late bound functions and their inverses[10], 2nd order functions [1], active rules[11]. AMOSQL furthermore has aggregation operators, nested subqueries, disjunctive queries, quantifiers, and is more than relationally complete.

AMOSQL provides a declarative access language to object databases. This declarativeness requires AMOSQL queries to be optimized before execution. The query optimizer in AMOS is cost-based and is optionally supporting three kinds of optimization methods:

- Optimization based on heuristics as in [1], called RANKSORT. This is the default optimization method which is very fast and normally works reasonably. However, occationally it will produce suboptimal plans.
- Optimization based on classical relational optimization as in [5], called EXHAUSTIVE, where exhaustive search is performed over all possible search plans to obtain the search plan with the lowest estimated execution cost. This gives an optimal execution plan but the method is NP-complete over the size of the query and can therefore only be used for small queries.
- Optimization based randomized sampling of favorable search plans from the space of possible search plans[4], called RANDOMOPT, developed from techniques devised by [3]. This method is slower than RANKSORT but not NP-complete as EXHAUSTIVE. It produces very good execution plans for both small and large queries, even though it does not always produce the optimal plan.

The optimizer is extensible whereby the user can provide cost formulae as AMOSQL functions to the optimizer for all three optimization methods. A generalized foreign function mechanism, *multi-way foreign functions* [1] is used to give transparent access to special purpose data structures such as the internal of the type system. By the foreign function mechanism the programmer can define AMOSQL functions in an external language (usually Lisp or C). The architecture relies on extensible optimization of such foreign function calls[1]. The use of the foreign function mechanism is described in the document **AMOS System Manual**.

The query compiler translates AMOSQL statements into an internal logic based language we call ObjectLog [1], which is an object-oriented dialect of Datalog [6]. As part of the translation into ObjectLog programs, some optimizations are applied on AMOSQL expressions relying on object-oriented properties of AMOSQL. In a second optimization step, the generated ObjectLog rules are transformed into an equivalent set of more efficient ObjectLog rules. The optimizer takes special care to optimize common object-oriented OSQL query patterns.

1.[See [2] for an overview of Iris and OSQL.

2.[In case the database is larger than the available main memory, the system still works but swapping may occur.

AMOS runs on several Unix platforms. The system includes a Lisp interpreter (subset of CommonLisp) which interprets a part of the system written in Lisp. The system has its own storage manager for the database, and a query language interpreter/compiler to process and optimize AMOSQL statements. To achieve good performance we have carefully optimized the representation of critical system data structures, e.g. the storage manager, object representation, type information, and the representation of function definitions. When advantageous, we use tailored main memory data structure representations, rather than using relations. For example, our object identifiers are represented as variable length records where one field points to the object's type information, and another points to its function definition. It is critical that this information is represented efficiently, since it is extensively looked up both during compilation and during interpretation of AMOSQL functions. The system uses around 800KB[1] of code and 800KB of meta data.

AMOSQL can call programs in regular programming languages, such as C and Lisp, and AMOSQL is also callable from these languages. The documentation of external programming language interfaces is documented in **AMOS System Manual**.

The remainder of this document describes the AMOSQL language. The features of AMOSQL are illustrated through a running example.

2 AMOSQL introduction

The basic concepts of the AMOS data model are *types*, *objects*, and *functions*. Objects are represented through typed atomic object identifiers (OIDs), and functions associate properties (attributes) with objects or define relationships between objects. Functions model object attributes and relationships between objects through four basic *function types*

- *Stored functions* are updatable tables that are stored in the database.
- *Derived functions* are side-effect free functions defined in terms of other AMOSQL functions. Derived functions are query optimized when they are defined.
- *Foreign functions* are defined using an external programming language (Lisp or C).
- *Database procedures* are programs in a procedural sublanguage of AMOSQL that may have side effects.

```
create type person;
create type student subtype of person;
create function Name(Person p) -> Charstring as stored;
create function Income(Person p) -> Integer as stored;
create function Bonus(Person p) -> Integer as stored;
create function Parent(Person c) -> bag of Person p as stored;
create function GrossIncome(Person p) -> Integer as
    select Income(p) + Bonus(p);
                   /* Derived where '+' is foreign */

create function SParent(Person c) -> bag of Student s as
    select Parent(c);    /* Parent if parent is student */


create function GrandSParentGrossIncome(Person c) -> bag of Integer gi as
    select gi                     /* Gross income of grandparent
                                     if grandparent is student */
    for each Person gp, Person p
    where GrossIncome(gp) = gi and
          SParent(p) = gp and
          Parent(c) = p;
```

1. On HP7xx workstations.

```
create function Income(Charstring nm) -> Integer as
    select Income(p)         /* Derived  overloaded */
           for each Person p
           where Name(p) = nm;
```

Figure 1 Examples of AMOSQL function definitions

Figure 1 shows examples of AMOSQL functions and types. Objects of type `Person` have the attributes `Income, Bonus, Parent, GrossIncome`, and `GrandSParentGrossIncome`. These are AMOSQL functions of a single argument bound to objects of type `Person`. Local variables are declared using the "`for each`" clause.

The operator `'+'` is internally represented by the AMOS function `Plus`. It is an example of a *foreign* function, implemented in C outside AMOSQL.

AMOSQL has a SQL-like `select` statement both for ad hoc queries to the database and for defining derived functions. The derived function `GrossIncome`, as any derived function, is defined through a single `select` statement that follow the "`as`" keyword.

The stored functions `Parent`, and the derived functions `SParent` and `GrandSparentGrossIncome` return sets (actually bags) of values.

In general, the arguments and results of a function together with their types are called the *signature* of the function. We denote signatures by

$f(T_1\ P_1,...,T_n\ P_n)$ -> $<U_1\ Q_1,...,U_m\ Q_m>$[1]
$P_1,...,P_n$ are the *arguments* of `f` whose values are of types $T_1,...,T_n$. A function can have as the result a set of tuples of values, $Q_1,...,Q_m$, with types $U_1,...,U_m$.

Overloaded functions are AMOSQL functions sharing a name for different definitions. In Figure 1there are two variants, or *resolvents*, of the overloaded `Income` function. Both resolvents provide a person's income, the first one is given an OID of an object of type `Person`, while the other is given a person's name as a string. Resolvents can be any of the four basic function types. AMOS chooses then the resolvent according to the argument type.

AMOSQL also provides *database rules* that are forward chaining rules that can monitor logical conditions and perform actions when these conditions become true.

3 Running AMOS

It is recommended that each user creates a private directory for AMOS, `<privdir>`. You should then do `cd <privdir>` and make the following two symbolic links from `<privdir>` to `<amosdir>`, the directory where the AMOS executables are stored:

```
ln -s <amosdir>/amos .
ln -s <amosdir>/amos.dmp .
```

AMOS is then ready to run in `<privdir>` by the command:

```
./amos [-W3] [<db> [<name>]]
```

where `[-W3]` is an option that enables a built in World-Wide-Web server and `[<db>]` is an optional name of an AMOS database image (default is `amos.dmp`) and `[<name>]` is an optional name given to the AMOS. If no `<name>` is given then the user name is used as default. The name is used when communicating with other AMOS servers. If a different name than the default name should be used, then `<db>` has to be supplied and must preceed `<name>`.

1.[The brackets around the result are optional for functions returning a single result.

The system then enters an *AMOS top loop* in which it reads AMOSQL statements, executes them, and prints their results. You need not connect to any particular database, but instead, if `<db>` is omitted, the system enters an empty database, where only the system objects are defined.

When the AMOS database is defined and populated, it can be saved on disk with the AMOSQL statement

```
save "filename";
```

NOTICE: Do not save using the name ’amos.dmp’ since it will overwrite the system database.

In a later session you can connect to the saved database by starting AMOS with

```
./amos filename
```

The AMOSQL statement

```
connect "filename";
```

connects to a saved database from the AMOS top loop.

The prompter in the AMOS top loop is

```
AMOS n>
```

where `n` is a *generation number*.

The generation number is increased every time an AMOSQL database update statement is executed. For example:

```
AMOS 1> create type person;
AMOS 2> create type student subtype of person;
```

Changes can be undone by using the `rollback` statement with a generation number as argument. For example, the statement:

```
AMOS 3> rollback 2;
```

will restore the database to the state it had at generation number `2`. It thus undoes the effect of the statement

```
create type student subtype of person;
```

After the rollback above, the type `student` is removed from the database, but not type `person`.

The statement `commit` makes changes non-undoable, i.e all updates so far cannot be rolled back any more and the generation numbering starts over from `1`. If the recovery system (“Recovery–Storage System” on page 37) is turned on the changes will also be saved on the disk.

If recovery is turned on the changes will be saved in a log file and be restored in case of a system crash.

For example:

```
AMOS 2> commit;
AMOS 1> ...
```

If persistency is turned off the `commit` does NOT write any data to disk; the `save` statement is used for that. A `rollback;` without arguments rolls everything back to the previous commit point. It is thus equivalent to `rollback 1;`.

To quit AMOS orderly first save the database and then type

```
AMOS 1> quit;
```

3.1 Running AMOS as World-Wide-Web server

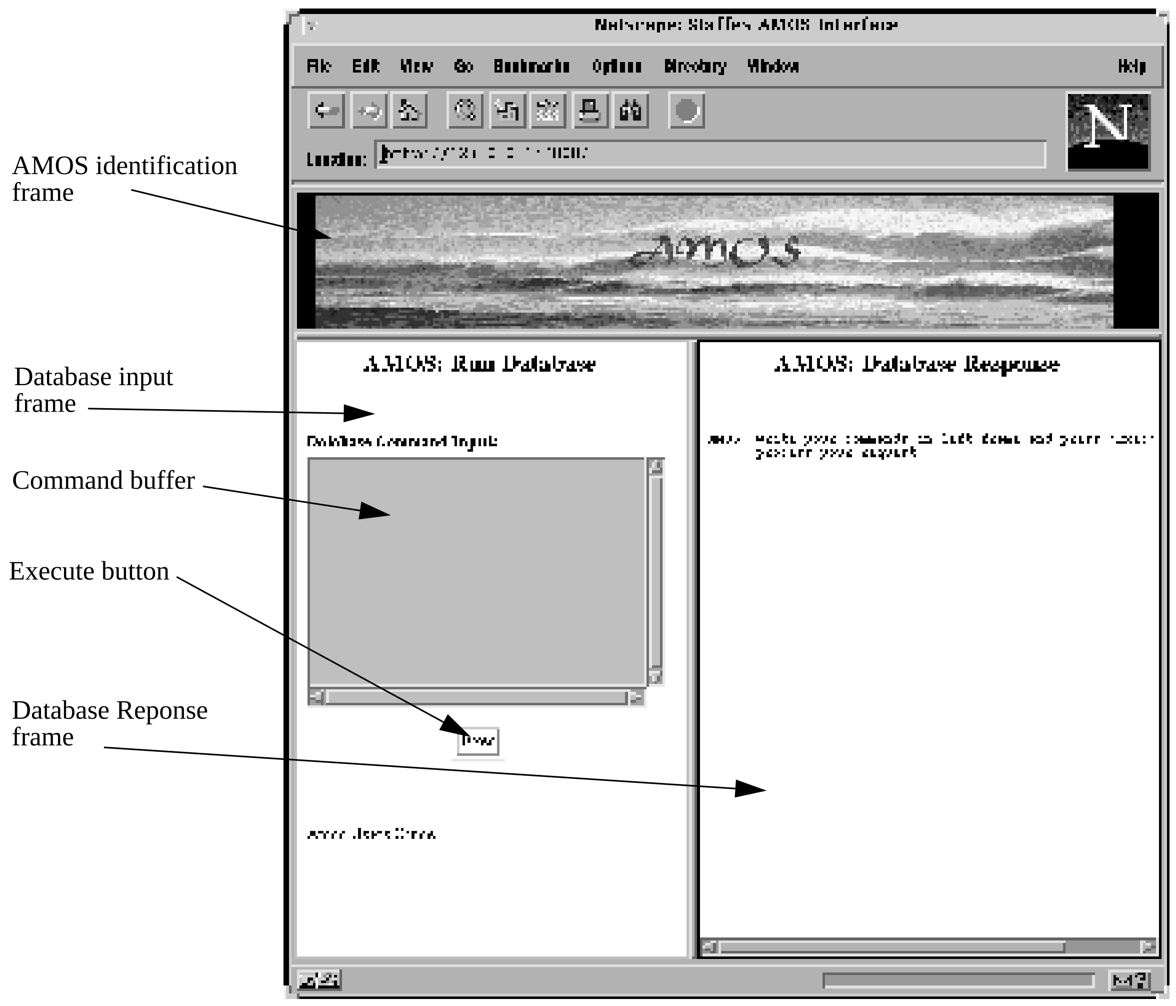
Integrated within AMOS is a W3 server. This server is enabled by starting AMOS using the -W3 option, see “Running AMOS” on page 6. By enabling the internal W3 server it is possible to use a web browser (e.g. Netscape, Mosaic, etc.) as user interface. The browser must be able to handle the HTML tag <FRAME>, e.g. Netscape V2. Whenever this W3 server is enabled any netscape client may connect to it. Thus, while this server is enabled the data in the database is open for public access.

When AMOS is started with the -W3 option it displays the port it is using to serve world wide web clients as:

```
mir31 <69> amos -W3
This is AMOS version unreleased
Copyright (c) EDSLAB, University of Linkoping, 1994
Based on WS-Iris, Copyright (c) HP, 1992
Listening to W3 socket AMOS W3 Server initialized, Using port 2000
AMOS 1>
```

In above example AMOS is using port 2000 for world wide web clients. The URL (Uniform Resource Locator) to use by the web browser is:’HTTP://<machine-name>:2000/’ where <machine-name> is the name or IP-address of the machine where AMOS is running. It is also possible to use the TCP/IP loopback address, e.g. 127.0.0.1 instead of the machine name if AMOS and the web browser is running on the same machine.

In the example above AMOS is started from the machine mir31, thus the URL is HTTP://mir31:2000/ and the result is pictured below.



The interface consists of three frames; *AMOS identification frame, Database input fram*e and *Database response* *frame*. To issue a command to AMOS type the AMOSQL command into the Command buffer in the database input frame and press the execute button. The response will then appear in the database response frame.

The database response frame is limited in size. The latest command will always appear in its full size but previous results may disappear after a while.

There are a few lisp functions defined which faciliate interactive control of the W3 server. These are:

`(start-www)`	Starts the www server if not already running
`(close-www)`	Closes the www server if it is running
`(status-www)`	Displays information about the current status of the wwwserver, i.e. if it is running or not and which port it uses.

NOTICE: The `'system()'` AMOSQL function is disabled when the www-server is running. This necessary in order to prevent unauthorized access to the host system. However.

4 General syntactic constructs

This next sections describe the statements available in AMOSQL. For the syntax we use BNF notation with the following special constructs:

`A ::= B C`: A consists of B followed by C.
`A ::= B | C`, alternatively (B | C): A consists of B or C.
`A ::= [B]`: A consists of B or nothing.
`A ::= B-list`: A consists of one or more Bs.
`A ::= B-commalist`: A consists of one or more Bs separated by commas.
`'x'`: The character 'x'. Useful for clarity or when syntax use reserved characters (i.e. `[, ],` and `|`).

AMOSQL statements are always terminated by a semicolon (`;`). All symbols are raised to upper case in AMOSQL statements, except within strings; i.e. AMOS ignores type cases in function and type definitions.

4.1 Identifiers

Identifiers have the syntax:

```
identifier ::=
  ('_' | letter) [identifier-character-list]
identifier-character ::=
  alphanumeric | '_'
```

Identifiers are NOT case sensitive; they are always translated to upper case internally[1].

4.2 Variables

Variables are of two kinds:

- *AMOSQL variables* are identifiers for data values in AMOSQL queries and functions. AMOSQL variables must be locally declared in function signatures ("Functions and Queries" on page 14), by `for each` clauses ("The select statement" on page 18), or by the `declare` construct ("Database procedures" on page 26).
  Syntax:
  `variable-name ::= identifier`
- *Interface variables* hold query results temporarily during a session and is also used to share values with foreign languages (C or Lisp) (see **AMOS System Manual**). Interface variables are global and cannot be referenced in function bodies.
  Syntax:

1.AMOS supports international language settings by checking the shell environment variable LANG (see local documentation for the library function setlocale).

```
interface-variable-name ::= ’:’ identifier
gen-variable-name ::= variable-name | interface-variable-name
```

4.3 Constants

Constants can be integers, reals, strings, booleans, or OIDs.
Syntax:

```
constant ::=
  integer-constant | real-constant | boolean-constant |
  string-constant | oid-constant
integer-constant ::=
  [’-’] digit-list
real-constant ::=
  [’-’] digit-list ’.’ [digit-list]
boolean-constant ::=
  TRUE | FALSE
string-constant ::=
  string-separator character-list string-separator
string-separator ::=
  ’’’ | ’"’
oid-constant ::=
  OID ’[’ 0x0, digit-list ’]’
```

In `string-constant`s the surrounding string separators must be the same. If the string contains the separator used, it must be preceded with the escape character backslash.

`oid-constant`s denote references to objects with a specified OID number. For example:

```
OID[0x0,1234]
```

OID constants allows you to get hold of any object in the system, but they should be avoided in AMOSQL programs!

4.4 Comments

The comment statement is a separator, i.e. it can be placed anywhere outside identifiers and constants.
Syntax:

```
comment ::=
  /* character-list */
```

5 Types

Types are represented as identifiers.
Syntax:

```
type-name ::= identifier
```

The `create type` statement creates a new user type. It also provides the option to create *attribute functions*. Attribute functions are represented as regular AMOSQL functions of a single argument. They can also be defined separately with `create function` statements (“Functions and Queries” on page 14).

Syntax:

```
create-type-stmt ::=
  CREATE TYPE type-name [SUBTYPE OF type-name-commalist]
        ['(' attr-function-commalist ')']

attr-function ::=
  function-name type-name [KEY]
```

For example:

```
 create type namedobject (name charstring key);
 create type person subtype of namedobject(age integer,
                                           parents bag of person);
create type student subtype of person (score integer);
```

Type names must be unique across all types.

The new type will be an immediate subtype of all the supertypes named in the `SUBTYPE` clause. The names in the `SUBTYPE` clause must be names of existing user types. If no supertypes are specified the new type becomes an immediate subtype of the system type `UserTypeObject`.

The `attr-function-commalist` clause is optional, and provides a way to create *attribute functions* on the new type. The attribute functions always have a single argument and a single result and are initially only *stored* [1] . The argument type is implicitly the type being created and the result type is specified by the `type-name`. The specified result types must denote existing types.

If `KEY` is specified for an attribute, it indicates that each value of the attribute is unique. ("Functions and Queries" on page 14).

5.1 System type hierarchy

The following types are predefined in AMOS:

```
AMOS
BAG OF x
BOOLEAN
CHARSTRING
CONTEXT
CURSOR
DATE
FUNCTION
INDEX
INTEGER
LIST
LITERAL
NUMBER
OBJECT
REAL
RULE
SAGA
TIME
TIMEVAL
```

1.[However, they can later be redefined as non-stored functions by separate `create function` statements.

```
TYPE
USERTYPE
USERTYPEOBJECT
VECTOR
```

“AMOS type hierarchy” on page 12 shows the upper levels of the type hierarchy of AMOS.

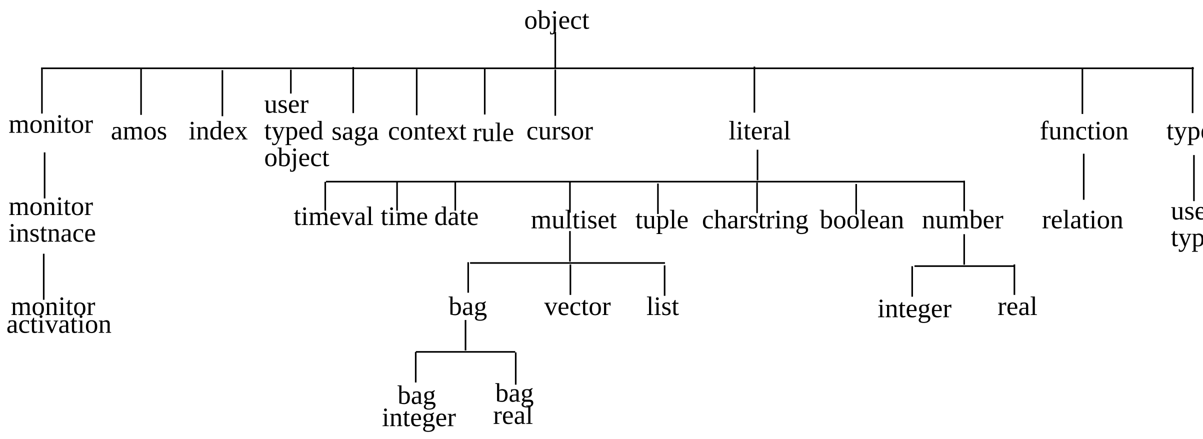


Figure 2 AMOS type hierarchy

- The root type is called `Object`.
- All user types are placed in the type hierarchy below `UserTypeObject`. The user is responsible for the structure of the type hierarchy below `UserTypeObject`.
- The type definitions are objects themselves and are instances of either the type `Type` for system types, or `UserType` for user types.
- A `literal` is a built-in type without explicit OID representation, usually representing basic programming language data types.
- The type **number** is a supertype covering both **integer** and **real** numbers.
- `Multisets` are literals holding collections of other values.
- Functions definitions are instances of the type `Function`.
- Cursors are instances ot the type `Cursor`.
- Rule definitions are instances of the type `Rule`.
- Rule contexts are used for dynamically grouping rules and are instances of type `Context`.
- Sagas are used for defining long-running transactions and are instances of type `Saga`.
- The type `bag` is a special data type that holds the result of queries as sets of objects with duplicates retained.
- `Amos` is a type used for handling distribution.
- `Timeval` is a type for absolute timestamps, `time` and `date` are types for relative points in time

5.2 Deleting types

The `delete type` statement deletes a type and all its subtypes.
Syntax:

```
delete-type-stmt ::=
  DELETE TYPE type-name
```

For example:

```
delete type person; /* Delete types person and student */
```

Functions using the deleted type will be deleted as well (“Functions and Queries” on page 14).

Objects that were instances of the deleted type will no longer be instances of that type, but they remain instances of other types they may have. All user objects are instances of `Usertypeobject`, so it is never the case that deleting a type causes objects to be left without any type.

6 Objects

The `create` statement creates one or more objects and makes the new object(s) instance(s) of a given user type and all its supertypes.
Syntax:

```
create-object-stmt ::=
  CREATE type-name
  ['(' function-name-commalist ')'] new-instances
new-instances ::=
  INSTANCES initializer-commalist
initializer ::=
  gen-variable-name |
  [gen-variable-name] '(' value-list-item-commalist ')'
value-list-item ::=
  simple-init-value | multiple-init-value | NIL
simple-init-value ::=
  single-value | multiset-value
single-value ::=
  gen-variable-name | constant
multiset-value ::=
  bag-value | vector-value
bag-value ::=
  'BAG(' single-value-commalist ')'
vector-value ::=
  'VECTOR(' single-value-commalist ')' |
  '{' single-value-commalist '}'
multiple-init-value ::=
  '<' simple-value-commalist '>'
```

Example:

```
create person (name,age) instances
  :adam ('Adam',26),:eve  ('Eve',32);
create person instances :olof;
create person (parents) instances
  :tore (bag(:adam,:eve));
```

The new objects are assigned initial values for the specified attribute functions. The attribute functions are any updatable AMOSQL functions.

One object will be created for each `initializer`. Each initializer can have an optional variable name which will be bound to the new object. The variable name can subsequently be used as a reference to the object.

The initializer also contains a comma-separated list of initial values for the specified functions. Initial values are specified as constants or variables.

The types of the initial values must match the declared result types of the corresponding functions.

Multiple result functions are initialized with a comma-separated list of values enclosed in angle brackets (syntax `multiple-value`).

Bag valued functions are initialized using the keyword `BAG` (syntax `bag-value`).

Vector result functions are initialized with a comma-separated list of values enclosed in curly brackets (syntax `vector-value`).

It is possible to specify `NIL` for a value when no initialization is desired for the corresponding function.

6.1 Object deletion

Objects are deleted with the `delete` statement.
Syntax:

```
delete-object-stmt ::=
  DELETE gen-variable-name
```

Example:

```
delete :adam;
```

The `delete` statement deletes a specified object from the database. Referential integrity is maintained by deleting references to the deleted object. It is thus also removed from all stored functions where it is referenced.

Deleted objects are printed as

```
OID[*DELETED*,1234]
```

The objects may be undeleted by rollback. The system garbage collects the OIDs from the database only when their creation has been rolled back or their deletion committed.

7 Functions and Queries

The `create function` statement creates a new user function that maps arguments of the specified argument types to results of the specified result types.
Syntax:

```
create-function-stmt ::=
  CREATE FUNCTION function-name argl-spec
         -> resl-spec [fn-implementation]
function-name ::=specific-function-name |
               type-name-list '.' specific-function-name '->' type-name-
list
type-name-list ::= type-name | type-name '.' type-name-list
specific-function-name ::= identifier
argl-spec ::='(' [arg-spec-commalist] ')'
arg-spec ::=simple-arg-spec | BAG OF simple-arg-spec
simple-arg-spec ::=type-name [variable-name] (KEY | NONKEY)
resl-spec ::=arg-spec | multiple-result-spec
multiple-result-spec ::='<' simple-arg-spec-commalist '>'
fn-implementation ::=AS (derived-body | procedure-body |
                         foreign-body | STORED)
derived-body ::= simple-select-stmt
foreign-body ::=FOREIGN [string-constant]
```

The `fn-implementation` specifies what the function does when it is invoked. The following kinds of function implementations are supported:

- A *stored function* (syntax `STORED`) represents data stored as facts in the database. The corresponding mapping between arguments and results are internally stored in a table. Stored functions can always be updated using the update statements (“Function Updates” on page 25). The set of mappings are initially empty.

- A *derived function* (syntax `derived-body`) is defined by a single AMOSQL query (`simple-select-stmt`) in the body. The query specifies how to map argument values to results. Queries are described below in “The select statement” on page 18 . The query optimizer is invoked when a derived function is defined so that optimization is not required when derived function are called (“Function Calls” on page 16). Some simple derived functions can be updated (“Function Updates” on page 25).
- A *foreign function* (syntax `foreign-body`) is defined by a program written in a foreign language (C or Lisp). The definition of foreign functions is documented separately in the document **AMOS System Manual**. Foreign functions can currently not be updated.
- A *database procedure* (syntax `procedure-body`) is a program written in a procedural subset of AMOSQL that may have side effects (i.e. database updates). Database procedures should not be used in queries. Database procedures are described in “Database procedures” on page 26 .
- An *overloaded function* is a function defined on different types with identical names. When a function call is made to an overloaded function, the appropriate implementation is selected based on the actual argument types. Overloaded functions are described in Sec. “Overloaded Functions and Late Binding” on page 19 .

The `argl-spec` and the `resl-spec` constrain the *argument* and the *result* types of AMOSQL functions. Examples:

```
create function age(person)->integer as stored;
create function income(person)->integer as stored;
create function richlimit()->integer as stored;
```

As the function richlimit illustrates, an AMOSQL function may have zero arguments.

Semantics:

- The types used in the declarations must be previously defined. The name of an argument or result parameter can be left unspecified if it is never referenced in the function implementation. The names of the argument and result parameters for a given function definition must be unique.
- `bag of` specifications on a single result parameter declares it to be a *bag*. A bag is a collection of objects, possibly with duplicates. For example:
  ```
  create function parents(person) -> bag of person;
  ```
  permits each person to have more than one parent. If `bag of` had been omitted each person could have only one parent (or none)[1]. This is a special case of a *cardinality constraint* which are described more in detail below.
- Derive functions can also have arguments declared `bag of`. Such functions are called *aggregation operators*. They are described in “Subqueries and Aggregation Operators” on page 21 .
  NOTICE: Stored functions cannot be aggregation operators.
- AMOSQL functions may also have *multiple results*, indicating that a logical tuple of values is returned. This is indicated by bracketing the result declarations (see syntax for `multiple-result-spec` and example below).

Derived AMOSQL functions are defined by a single query (or `select` statement). The syntax and semantics of queries is explained in “The select statement” on page 18). Some examples of derived functions:

```
create function double_income (person p)->integer
  as select income(p) + income(p);
create function rich(person p)->boolean
  as select where income(p) > richlimit();
create function Married(Person h,Person w) -> Boolean as stored;
create function Wife(Person h)->Person w
```

1.However, in the current implementation bag-valued stored functions can not return duplicates, i.e. stored functions are always set valued. Derived functions can however be bag-valued.

```
      as select w where Married(h,w);
    create function family_incomes(person p) -> <integer h,integer w>
      as select income(p),income(wife(p));
     create function family_totalincome(person p)->integer t
      as select hi + wi
      for each integer hi,integer wi
      where <hi,wi> = family_incomes(p);
```

The function `wife` is an example of a derived function calling a boolean function.
The function `family_incomes` is an example of a multiple result derived function.
The function `family_totalincome` is an example of a derived function calling a multiple result function.

7.1 Function Calls

The simplest form of a query is to make a *function call*. Syntax:

```
    function-call ::=
      function-name '(' [parameter-value-commalist] ')'
    parameter-value ::=
      function-call | single-value | '(' simple-select-stmt ')'
    simple-select-stmt ::=
      SELECT expr-commalist
      [for-each-clause] [where-clause]
    expr   ::=
      function-call | single-value
```

For example:

```
     age(:eve);
```

The above function call is equivalent to the ad hoc query

```
     select age(:eve);
```

The query optimizer is not invoked for such simple unnested calls to AMOS functions and the invocation is very fast. The reason is that the query optimizer is instead applied when a derived function is defined, and the function is optimized for unnested calls to the function.

Function calls can also be nested, for example:

```
     age(parents(:eve));
```

or equivalently

```
     select age(parents(:eve));
```

For nested function calls the query optimizer will be applied to produce an execution plan for the call, which is then immediately executed. Thus nested function calls are significantly slower than unnested ones, and the user is recommended to avoid nested function calls by defining suitable derived functions. In the example above, define

```
     create function parage(person p) -> integer
      as select age(parents(p));  /* Optimizer invoked here
                                    execution plan saved in db */
    parage(:eve); /* Optimizer NOT invoked here */
```

Daplex semantics is used for nested function calls. This means that if a function is applied on an inner function which is bag valued (such as `parents`) the outer function (`age` above) is applied on each elements of the bag. In the example above, the result of the query `parage(:adam);` is a bag of the ages of all parents of `:adam`.

The Daplex semantics is NOT used for `aggregation operators` (See “Subqueries and Aggregation Operators” on page 21) which are functions that aggregate over the bag. For example, to count how many parent `:adam` has, the aggregation operator `count` can be used which counts the elements of its argument bag:

```
count(parents(:eve));
```

The built-in functions `+,-,*,/` have equivalent infix syntax with the usual priorities. For example:

```
(income(:eve) + income(:ulla)) * 0.5;
```

is equivalent to

```
times(plus(income(:eve),income(:ulla)),0.5);
```

7.2 Cardinality Constraints

A *cardinality constraint* is a system maintained restriction on the number of occurrences of a function parameter. For example, a cardinality constraint could be that there is at most one salary and name per person, while a person may have any number of parents. The only cardinality constraint that is currently supported in AMOSQL is to make a given parameter of a stored function unique, i.e. a given object can participate at most once as actual argument or result of the function. This is done by attaching the keyword `KEY` immediately after a parameter declaration. For example:

```
create function name(person key) -> charstring key as stored;
```

Indicates that there is only one name for each person and that the names of persons are unique.
If the `KEY` cardinality constraint is violated by a database update the following error message is printed:

```
Update would violate upper object participation (updating function ...)
```

The keyword `NONKEY` specifies that the parameter has no cardinality constraint.
The default cardinality constraint is `KEY` for the first argument of a stored function and `NONKEY` for all others. This implies that stored functions are by default single valued.

The `name` function could thus also have been written:

```
create function name(person) -> charstring key as stored;
```

For foreign functions it is up to the implementor to guarantee that specified cardinality constraints hold.
Cardinality constraint declarations are ignored for derived functions.

For example:

```
 create function Married(Person husband,Person wife key)->Boolean as
stored;
```

Polygamous marriages are refused to be stored by the function `Married`, since the first argument has the default cardinality constraint `KEY`.

The `bag of` declaration on the result of a stored function actually just overrides the default `KEY` declaration of its argument with `NONKEY`. Thus the function `parents` above could also have been written:

```
create function parents(person nonkey) -> person as stored;
```

7.3 Deleting functions

Functions are deleted with the `delete function` statement.

Syntax:

```
delete-function-stmt ::=
  DELETE FUNCTION function-name
```

For example:

```
 delete function married;
```

Deleting a type also deletes all subtypes and all functions using the deleted types.

For example:

```
 delete type namedobject; /* Wipes out all functions and
                              types defined so far */
```

7.4 The `select` statement

Queries retrieve objects having specified properties. They are specified using the `select` statement.

Syntax:

```
query-stmt ::=
  select-stmt | function-call
select-stmt ::=
  SELECT[DISTINCT] expr-commalist
        [into-clause]
        [for-each-clause]
        [where-clause]
into-clause ::=
  INTO gen-variable-name-commalist
for-each-clause ::=
  FOR EACH variable-declaration-commalist
variable-declaration ::=
  type-name variable-name |
  BAG OF type-name variable-name
where-clause ::=
      WHERE predicate-expression
```

For example,

```
select age(p) for each person p where name(p) = 'Eve';
```

Selects the age of the person named Eve.

```
 select name(q) for each person p, person q
  where name(p) = 'Eve' and age(q) > age(p);
```

selects the names of all persons older than Eve.

Function calls (“Function Calls” on page 16) is the simplest form of queries.

The `expr-commalist` defines the object(s) to be retrieved. See “Predicate expressions” on page 19 for definition of `function-call`.

The `for-each-clause` declares types of local variables used in the query.

The `where-clause` gives a selection criteria for the search. The details of the where clause is described below in “Predicate expressions” on page 19 .

The result of a select statement is a *single result tuple* of objects or a *bag* of single result tuples. Duplicates may occur, unless suppressed using the DISTINCT keyword.

An `into-clause` is available for specifying variables to be bound to the result. In case more than one result tuple is returned, the variables will be bound only to the elements of the first encountered tuple. For example:

```
select p into :eve2 for each person p where name(p) = ’Eve’;
name(:eve2);
```

This query retrieves into the environment variable `:eve2` the person whose name is ’Eve’.

7.5 Predicate expressions

The general syntax of predicate expressions is:

```
predicate-expression ::=
  predicate-expression AND predicate-expression |
  predicate-expression OR predicate-expression |
  ’(’ predicate-expression ’)’ |
  simple-predicate
simple-predicate ::=
  function-call |
  relterm relop relterm
relterm ::=
  function-call | res-values
res-values ::=
  single-value | ’<’ single-value-commalist ’>’
relop ::=
  = | < | > | <= | >= | !=
```

In a function call, the types of the actual parameters and results must be the same as, or subtypes of, the types of the corresponding formal parameters or results.

Resolution of overloaded functions is described in“Overloaded Functions and Late Binding” on page 19.

Query variables can be bound to bags. The treatment of bag variables and aggregation operators is described in “Subqueries and Aggregation Operators” on page 21 .

The comparison operators (`=, !=, <, <=, >` and `>=`) are treated as binary boolean functions. They are defined for any object type.

7.6 Overloaded Functions and Late Binding

Function names may be *overloaded*, i.e., functions having the same name may be defined differently on different argument types. This allows generic functions to apply to several different object types. Each specific implementation of an overloaded function is called a *resolvent*.

For example, assume the two following AMOS function definitions:

```
create function less(number i, number j)->boolean
  as select where i < j;
create function less(charstring s,charstring t)->boolean
  as select where s < t;
```

Its resolvents will have the signatures:

```
less(number,number) -> boolean
less(charstring,charstring) -> boolean
```

Internally the system stores the resolvents under different function names. The name of a resolvent is obtained by concatenating the type of its arguments with the name of the overloaded function followed by the symbol ’`->`’ and the type of the result.(syntax in “Functions and Queries” on page 14). The two resolvents above will be given the names `number.number.less->boolean` and `charstring.charstring.less->boolean`.

Overloaded function resolvents are allowed to differ on their argument types and the result types. The query compiler resolves the correct resolvent to apply. If there is an ambiguity, i.e. several resolvents qualify, or if no resolvent qualify an error will be generated by the query compiler.

When overloaded function names are encountered in AMOSQL function bodies, the system will use local variable declarations to choose the correct resolvent (early binding). For example,

```
create function younger(person p,person q)->boolean
  as select less(age(p),age(q));
```

will choose the resolvent `number.number.less->boolean`, since `age` returns integers and the resolvent `number.number.less->boolean` is applicable to `integers` by inheritance. The other function resolvent `charstring.charstring.less->boolean` does not qualify since it is not legal to apply to arguments of type `integer`.

On the other hand,

```
create function nameordered(person p,person q)->boolean
  as select less(name(p),name(q));
```

will choose the resolvent `charstring.charstring.less->boolean`. In both cases the resolution will be done at compile time.

Dynamic type resolution is also done for top level function call to choose the correct resolvent. For example,

```
less(1,2);
```

will choose `number.number.less->boolean`

NOTICE: To avoid the overhead of dynamic type resolution one may use the ’dot notation’:

```
number.number.less->boolean(1,2);
```

AMOS supports also *late binding* of overloaded functions where the overload resolution is done at run time instead of at compile time. For example, suppose that managers are employees whose incomes are the sum of the income as a regular employee plus some manager bonus:

```
create type employee subtype of person;
create type manager subtype of employee;
create function mgrbonus(manager)->integer as stored;
create function income(employee)->integer as stored;
create function income(manager m)->integer i
  as select employee.income(m) + mgrbonus(m);
```

Now, suppose that we need a function that returns the gross incomes of all persons in the database, i.e. we use `manager.income` for managers and `employee.income` for non-manager. In AMOS such a function is defined as

```
create function grossincomes()-> integer i
  as select income(p)
  for each employee p; /* income(p) late bound */
```

Since income is overloaded with resolvents `employee.income->integer` and `manager.income->integer` and both qualify to apply to employees[1], the resolution of `income(p)` will be done at run time. If income of employees are sought the desired resolvent has to be explicitly specified as `employee.income->inte-`

1.Due to inclusion polymorphism, Objects of type manager are also of type employee `ger`.

We currently have the restriction on late bound functions that they can only be used in the `forward` direction, i.e. queries where their arguments are known but not their results.

Since the detection of the necessity of dynamic resolution is done at compile time, overloading a function name may lead to a cascading recompilation of functions defined in terms of that function name. This can take some time. For a more detailed and extensive presentation of the managemant of late bound functions see [9][10].

7.7 Disjunctive Queries

The OR operator works like a union operator, i.e. the union of the objects satisfying its operands (without duplicates removed) is returned. Queries and function definitions can have arbitrary nesting of ANDs and ORs.

Example:

```
create function father(person) -> person as stored;
create function mother(person) -> person as stored;
create function parent(person p) -> person q
  as select q where q=father(p) or q=mother(p);
```

The function body of `parent` is a *disjunctive query*, since it contains an OR. `parent` would generate the bag of all fathers and mothers for a given person.

7.8 Subqueries and Aggregation Operators

Normally when an AMOSQL function is applied on a bag-valued function it is applied on each element of the bag. For example,

```
age(parents(:eve));
```

will return the ages of all parents of `:adam`, i.e. the function `age` is applied on each element of the result from the function call `parents(:adam)`.

By contrast, an aggregation operator is a function that treats some bag valued argument(s) as a single unit. Thus the complete bag is passed at once to the outer function, rather than applying the outer function on each result of the bag. For example, `count(bag of object)` is an example of an aggregation operator:

```
 count(parents(:eve));
```

In this case the number of parents of `:adam` is returned.

Aggregation operators are defined as functions where one or several arguments are declared as bags:

```
      bag of type x
```

The following system aggregation operators are defined:

```
sum(bag of integer x) -> number r
sum(bag of real x) -> number r
count(bag of object x) -> integer r
maxagg(bag of object x) -> object r
minagg(bag of object x) -> object r
```

NOTICE: Aggregation operators can be overloaded (as for `sum`). Also notice that `count, maxagg` and `minagg` can be applied to any bag, while `sum` must be applied only to 'uniform' bags of integers or reals.

NOTICE: AMOS supports nested subqueries as arguments to aggregation functions. For example, `totalin-`

comes could also be written:

```
create function totalincomes()->integer
  as select sum((select i
                     for each person p,integer i
                     where income(p)=i));
                /* income will be bound at runtime */
```

Subqueries always return bags as their result; thus the result of a subquery must be passed to only aggregation operators.

NOTICE: Nested subqueries must syntactically be enclosed in parentheses, as in `totalincomes` above.

Local variables in queries may be declared as bags. For example `totalincomes` could also have been written:

```
create function totalincomes()->integer
  as select sum(b) for each bag of integer b
  where b = (select i for each person p,integer i
                where income(p)=i);
         /* Late binding on income*/
```

7.9 Negated subqueries and quantification

There are two aggregation operators in AMOSQL to test if a bag is empty, `notany`, or not empty, `some`:

```
notany(bag of object x) -> Boolean
some(bag of object x)-> Boolean
```

For example, the function `butlowestincomes(d)` returns all incomes that is more than `d` higher than the lowest income:

```
create function butlowestincomes(integer d)->integer i
  as select i for each integer i,person p
  where some((select for each person q
                where (income(q) + d) < i))/* Late */
                     and i = person.income(p);
```

`some` corresponds to the logical quantifier **exists**.

Contrast this to the complementary function `lowestincomes` that computes all incomes within the distance `d` from the lowest of all incomes:

```
create function lowestincomes(integer d)->integer i
  as select i for each integer i,person p
  where notany((select for each person q
                     where (income(q) + d)<i))/* Late */
                           and i = person.income(p);
```

`notany` corresponds to the logical quantifier `not exists`, i.e. it negates subqueries, while `some` corresponds to `exists`.

7.10 Transitive Closures

A transitive closure is all objects, o, reached directly or indirectly from an object, s, by applying some function, f. The classical example is to find all ancestors of a given person following the `parent` function (or finding all subparts of a given part).

The recommended way to compute transitive closures in AMOS is to use the built-in function `tclose`:

```
tclose(function f,object o,integer maxdepth)-> <object r,integer depth>
```

Starting with object `o` it constructs the transitive closure by successively applying `f(o), f(f(o))` etc. down to level `maxdepth`. `tclose` returns the objects, `r`, in the closure and their distance, `d`, from `o`. `f` must be function with a single argument and result.

`tclose` is overloaded so that, as an alternative, the name of the traversal function can be specified as a string.

Example:

```
create function ancestors(person o)-> bag of person a
  as select a for each integer d
  where  tclose("person.parents->person",o,200) = <a,d>
                   and a != o;
```

The `tclose` function is invertible if the traversal function is invertible. This means that the direction of the transitive closure can be inverted. Thus both these queries are legal:

```
ancestors(:kain);
select p for each person p where ancestors(p) = :eve;
```

The first query (function call) returns all ancestors of `:kain` while the other query returns all descendants of `:eve`[1].

An alternative definition of `ancestor` is as a recursive function (see next section). However, it is recommended to use `tclose` as an alternative to recursion whenever possible, since this is more efficient. Studies have shown that transitive closures account for a large majority of the needs for recursive queries. Since `tclose` is invertible in our case, it will be almost as powerful as the recursive definition.

7.11 Cursors

For queries and function calls returning bag valued results, the `open-cursor-stmt` and the `fetch-cursor-stmt`, statements are available to iterate over the result.
Syntax:

```
open-cursor-stmt ::=
  OPEN cursor-name FOR query-stmt
cursor-name ::=
  gen-variable-name
fetch-cursor-stmt ::=
  FETCH cursor-name (into-clause | next-clause)
next-clause ::=
  NEXT integer-constant
close-cursor-stmt ::=
  CLOSE cursor-name
```

For example:

```
create person (name,age) instances :Viola ('Viola',38);
open :c1  for select p for each person p where name(p) = 'Viola';
fetch :c1 into :Viola1;
close :c1;
name(:Viola1);
<"Viola">
```

A cursor is created by the `open-cursor-stmt` and is represented by a cursor object of *result tuples* containing objects with unknown types.

The result of the query is materialized if it doesn't exceed 1000 tuples. Should the number of tuples be greater then

1.[I.e. the set of all humans that ever lived(except two).

the result isn't fully materialized but will be created as it is retrieved.

Construction of cursor contents when the result is greater than 1000 tuples is handled by cursor processes. Should a cursor process die unexpectedly an error message is issued and the interface variable holding the cursor becomes unbound. However, this check is made only when cursors are opened or closed for efficiency reasons. Therefore, it might be possible to retrieve a partial result, with `fetch`, from a cursor whose cursor process died unexpectedly. Also, should the system be unable to create a cursor process, an error message saying so will be issued.

The `fetch-cursor-stmt` fetches the first result tuple(s) from the cursor; i.e. the tuple(s) is removed from the front of the cursor bag. NIL is returned if there are no more result tuples left in the cursor.

If present in a `fetch-cursor-stmt`, the `into` clause will bind elements of the first result tuple to AMOSQL interface variables. There must be one interface variable for each element in the result tuple.

If present in a `fetch-cursor-stmt`, the `next` clause will display the specified number of result tuples and remove them from the cursor bag.

If neither a `next` nor an `into` clause is present in a `fetch-cursor-stmt`, a single result tuple is fetched and displayed.

The `close-cursor-stmt` removes the cursor.

When a `commit` is done all open cursors are closed.

It is sometimes useful to count the number of result tuples in a cursor bag:

```
create function ageofpersonnamed(charstring nm)-> integer a
  as select age(p) for each person p where name(p)=nm;
open :c1 for call ageofpersonnamed('Eve');
count(:c1);
<1>
```

This possibility should be used with great care since a complete materialization of the cursor is performed. Should the cursor contain a large result then memory might become exhausted.

7.12 Recursive functions

AMOS supports a limited class of recursive queries. Recursive functions are normally disjunctive.
For example:

```
create function ancestors(person p)->person a
  as select a for each person q
  where (a = ancestors(q) and q = parent(p)) or a = parent(p);
```

Recursive queries are evaluated top-down. The system only handles recursive functions that call themselves recursively in the 'forward' direction (where all arguments are known). Otherwise, the system will complain that the query is not executable ('unsafe'). Left recursive functions, as `ancestor` are re-ordered by the optimizer to become right-recursive (in order to avoid internal looping). The top-down evaluation may still cause indefinite looping in case there are circularities in the data.

NOTICE: Because of the above problems it is recommended to use the transitive closure function (see "Transitive Closures" on page 22) as an alternative to recursion whenever possible. Extra care should be taken when defining overloaded recursive function as the possibility of infinite loops exists.

8 Database updates

We describe how to update the contents of the database. Notice that database population by object creation and attribute assignments was described in"Objects" on page 13 .

8.1 Function Updates

Information in AMOSQL can be thought of as mappings from function arguments to results. These mappings are either defined at object creation time (“Objects” on page 13), or altered by one of the *function update statements* SET, ADD, or REMOVE.
Syntax:

```
update-stmt ::=
  update-op update-item [for-each-clause] [where-clause]
update-op ::=
  SET | ADD | REMOVE
update-item ::=
  function-name ’(’ single-value-commalist ’)’ ’=’ res-values
```

Not every function is updatable. AMOS defines a function `f` to be updatable if it is a stored function, or if it is derived from a single updatable function `g` in such a way that the argument and result parameters of `f` partition all the arguments and results of `g` and such that no selection is involved in the derivation.

Semantics:

`set` sets the value of an updatable function given the arguments. For example:

```
set age(:adam)=33;
```

`set` can be combined with querying for set-oriented updates. For example:

```
set age(p)= q for each person p, integer q
  where q = 1 + age(p);
```

will iterate over all persons and increment their ages.

A boolean function can be set to either TRUE or FALSE.

`add` adds the specified tuple(s) to the result of an updatable bag result function, analogous to `set`.

`remove` removes the specified tuple(s) from the result of an updatable bag result function, analogous to `set`.

The update statements are not allowed to violate the cardinality constraints (KEY) (See “Functions and Queries” on page 14.) specified by the `create-type-stmt` or the `create-function-stmt`.

8.2 Updating type memberships

The `add-type-stmt` updates the type membership of one or more objects to make it belong to the specified type.

Syntax:

```
add-type-stmt ::=
  ADD TYPE type-name [’(’ [function-name-commalist] ’)’]
  TO new-instances
```

The updated objects may be assigned initial values for all the specified attribute functions in the same manner as in the `create object` statement.

The `remove-type-stmt` makes one or more objects no longer belong to the specified type.
Syntax:

```
remove-type-stmt ::=
  REMOVE TYPE type-name FROM variable-name-commalist
```

Referential integrity is maintained so that all references to the objects as instances of the specified type cease to exist.

An object will always be an instance of some type. If all user defined types have been removed, the object will still be member of `UserTypeObject`.

9 Database procedures

A database procedure is an AMOS function defined as a sequence of AMOSQL statements that may have side effects (i.e. database update statements or variable assignments). Procedures may return results by using a special `result` statement. Procedures should **not** be used in queries (but this restruction is currently not enforced). Most, but not all, AMOSQL statements are allowed in procedure bodies as can be seen by the syntax below.

Syntax:

```
        procedure-body ::=
            block |
            create-type-stmt |
            create-object-stmt |
            create-function-stmt |
            create-rule-stmt |
            create-index-stmt |
            delete-type-stmt |
            delete-object-stmt |
            delete-function-stmt |
            delete-rule-stmt |
            delete-index-stmt |
            for-each-stmt |
            update-stmt |
            set-variable-stmt |
            fetch-cursor-stmt |
            open-cursor-stmt |
            close-cursor-stmt |
            select-stmt |
            if-stmt |
            result-stmt |
            activate-rule-stmt |
            deactivate-rule-stmt |
            quit-stmt
    block ::=  BEGIN procedure-body-semicolonlist END |
            BEGIN
            DECLARE variable-declaration-commalist ’;’
            procedure-body-semicolonlist
            END
        result-stmt ::=
            RESULT expr
        for-each-stmt ::=
            FOR EACH [DISTINCT] variable-declaration-commalist
                    [where-clause] procedure-body
        if-stmt ::=
            IF predicate-expression
            THEN procedure-body
            [ELSE procedure-body]
        set-variable-stmt ::=
            SET gen-variable-name ’=’ expr
```

Examples:

```
create function creperson(charstring nm,integer inc) -> person p
               as
               begin
                 create person instances p;
                 set name(p)=nm;
                 set income(p)=inc;
                 result p;
               end;

set :p = creperson('Karl',3500);
create function makestudent(object o,integer sc) -> boolean
               as add type student(score) to o (sc);
makestudent(:p,30);
create function flatten_incomes(integer threshold) -> boolean
               as for each person p where income(p) > threshold
                    set income(p) = income(p) -
                                    ((income(p) - threshold) / 2);
flatten_incomes(1000);
```

A procedure is a function with one or several *procedural statements* in its body. The `block` construct is used to store several procedural statements in the body.

Procedures are compiled at definition time.

Procedures may return (bags of) results. The `result-stmt` is used for this, where the form is evaluated and returned as the result from the procedure.

The `for-each-stmt` construct can be used to iterate over the result of a query. For example the following function adds `inc` to the incomes of all persons with salaries higher than `limit` and returns their *old* incomes:

```
create function increase_incomes(integer inc,integer limit)
                                        -> integer oldinc
           as for each person p, integer i
              where i > limit
                and i = income(p)
             begin
               result i;
               set income(p) = i + inc
             end;
```

NOTICE: The semicolon can be omitted after last statement in a block.

NOTICE: `result-stmt` does not not change the control flow (different from, e.g., `return` in C), but it only specifies that a value is to be added to the result bag of the function and then the procedure evaluation is continued as usual. The `for-each-stmt` does not return any value at all unless `result-stmt` is used within its body.

NOTICE: Queries and updates embedded in procedure bodies are optimized at compile time. The compiler saves the optimized query plans in the database so that dynamic query optimization is not needed when procedures are executed.

10 Database rules

Rules have been introduced in AMOSQL. These can be used as integrity constraints that abort or compensate for inconsistent updates. Rules can also be used as a way for applications to monitor specific events in the database.

The syntax for rules conforms to that of AMOSQL functions as closely as possible:

```
create-rule-stmt ::=
  CREATE RULE rule-name param-spec AS
  [variable-declaration-commalist]
  WHEN predicate-expression
  DO procedure-body
```

Rules are deleted by:

```
delete-rule-stmt ::=
  DELETE RULE rule-name
```

The *predicate-expression* can contain any boolean expression, including conjunction, disjunction and negation. Rules are activated and deactivated by:

```
activate-rule-stmt ::=
  ACTIVATE RULE rule-name ([parameter-value-commalist])
  [PRIORITY (0|1|2|3|4|5)][STRICT] [INTO context-name]

deactivate-rule-stmt ::=
  DEACTIVATE RULE rule-name ([parameter-value-commalist]) [FROM context-
name]
```

The semantics of a rule are as follows: If an event of the database changes the boolean value of the condition to *true*, then the rule is marked as *triggered*. If something happens later in the transaction which causes the condition to become false again, the rule is no longer triggered. This ensures that we only react to logical events[1] . In the *check phase* (usually done before committing the transaction), the actions are executed of those rules that are marked as triggered. If an action is to be executed only once per activation, the rule can be deactivated as the last instruction in the rule action. By using priorities at rule activation the user can specify the order of rule execution in case of simultaneous triggering of several rules. Rules have 'nervous' semantics as default. This means that a rule will trigger everytime that a condition becomes *true* even if it was already *true*. By adding the keyword *strict* at rule activation the rule will have 'strict' semantics which is defined as: If an event of the database changes the boolean value of the condition from *false* to *true*, then the rule is marked as *triggered*. Rule context specifies if a rule is to be activated into a specific rule context or be deactivated from a specific context. The default is the `deferred` rule context, where rules are automatically checked at (or actually just before) transaction commit. See section 10.1 for more about contexts.

Example 1:

The salary changes of employees and managers are to be monitored. We want to ensure that only managers can have their salaries reduced. First we define the employee and manager types and the respective income functions, where managers receive an additional bonus:

```
create type person;
create type employee subtype of person;
create type manager subtype of employee;
create function name(person) -> charstring as stored;
create function mgrbonus(manager) -> integer as stored;
create function income(employee) -> integer as stored;
create function income(manager m) -> integer i
  as select i where i = employee.income->integer(m) + mgrbonus(m);
create employee(name,income) instances
```

1.To support physical events the system should provide functions that change values whenever a physical event occurs and thus can be referenced in the condition of a rule.

```
    :joe ('Joe Smith',30000);
  create manager(name,employee.income->integer) instances
    :harold ('Harold Olsen',80000);
  set mgrbonus(:harold) = 10000;
```

Define a procedure for updating the income:

```
  create function previous_income(employee) -> integer as stored;
  create function set_income(employee e, integer i) -> boolean
    as
    begin
    set previous_income(e) = income(e);
    set income(e) = i;
    end;
```

Then we define procedures for what to do when a salary is decreased:

```
  /* employee income cannot be decreased */
  create function compensate(employee e) -> boolean
    as set income(e) = previous_income(e);
  /* dummy procedure, managers are not compensated */
  create function compensate(manager) -> boolean;
```

Finally we define the rule to detect decreasing salaries for all employees:

```
  create rule no_decrease() as
    for each employee e
    when income(e) < previous_income(e)
    do compensate(e);
```

Activate the rule:

```
  activate rule no_decrease();
```

If an employee that is not a manager gets his salary decreased, the rule will automatically set the salary back to the old value at check time:

```
  set_income(:joe, 20000);
  check(); /* => reset income(:joe) to 30000 */
```

Commit does an implicit check:

```
  set_income(:joe, 25000);
  commit; /* => reset income(:joe) to 30000 */
```

Note: Since the rule is defined for all employees, and manager is a subtype of employee, the rule is overloaded for managers. (Because the functions `income` and the procedure `compensate` are overloaded). If a person of type manager gets a salary reduction, no action is taken. This is an example of a set-oriented rule. The action is executed for every binding of the universally quantified variable `e` for which the condition is true.

Example 2:

Rules can be parameterized and instantiated with different arguments. Take a rule that ensures that a specific employee has an income below a certain maximum income, and the transaction is rolled back if an employee receives an income above the threshold. This maximum income is fixed for all employees, but can vary for individual managers.

```
create function maxincome(employee) -> integer
  as select 50000;
create function maxincome(manager) -> integer as stored;
create rule exceeding_maxincome(employee e) as
  when income(e) > maxincome(e)
  do rollback;
```

Set the income limit for Harold:

```
set maxincome(:harold) = 120000;
```

Activate the rule for a particular employee Joe and manager Harold:

```
activate rule exceeding_maxincome(:joe);
activate rule exceeding_maxincome(:harold);
set income(:joe) = 75000; /* rollback at check time because 75000 > 50000
*/
set maxincome(:harold) = 80000;
  /* rollback at check time because 80000 + 10000 > 80000 */
 set mgrbonus(:harold) = 45000;
  /* rollback at check time because 80000 + 45000 > 120000 */
```

10.1 Rule contexts

Rule contexts [12] are a mechanism for dynamically grouping rules. An application can have several contexts that it activates and deactivates at different times. When rules are activated in AMOS, they are always associated with rule contexts. Only activated rules in activated contexts are monitored during a transaction. The contexts are first-class objects and are created by the statement:

```
CREATE context-name
```

where the *context-name* is a global name. Contexts are deleted by:

```
DELETE CONTEXT context-name
```

The contexts are initially *inactive* which means that before a context is *activated* the events affecting its rules are not monitored (unless the events are monitored by another already active context). Contexts are activated by:

```
ACTIVATE CONTEXT context-name
```

which enables all the activated rules in the context to be monitored. Contexts are deactivated by:

```
DEACTIVATE CONTEXT context-name
```

which disables all the activated rules in the context from being monitored. Two built-in contexts, named `deferred` and `detached`, are predefined and always active for deferred and detached rules, respectively. These are checked automatically by the system. Deferred rules are checked immediately before transaction-commit and detached immediately after. Rules are objects that can be fetched with the function `contextnamed` that takes the name as a charstring and returns the context object. The active rules in a context are checked by calling the function `check` with the context object as argument. Note that the context itself must be active as well, otherwise the check operation will have no effect.

11 Sagas for long-running transactions

The AMOS transaction system has been extended with sagas. Sagas are first class objects and can be used to chain a sequence of committed transations with compensating transactions. The sequence of sagas can be nested by defining sub-sagas. Abortion of a saga causes all the compensations to be executed and the sagas (and sub-sagas) to be deleted. Committing a saga just causes deletion (since the transactions are already committed). Compensation of one saga is done in one complete sequence (unless stopped). If an application needs to schedule sagas (forward and backward) in smaller steps it is possible to orchestrate many sagas through a saga layer (as part of the application) outside AMOS.

Sagas are created by the following function calls:

```
set :s = create_saga();
```

or

`set s = create_sub_saga();` (only to be used within another saga)

The syntax for executing something in a saga is as follows:

```
saga-stmt ::=
      SAGA saga procedure-body
      COMPENSATION procedure-body
```

Sagas are commited (and deleted) by:

```
commit_saga(:s);
```

and are aborted (and deleted) by

```
abort_saga(:s);
```

During abortion of a saga all the compensations are executed until the beginning or until stopped by a call to

```
stop_compensation();
```

Sagas can be passed to procedures to be executed in the body of the procedure. Note that any local variables defined outside the saga statement will have the values in the compensation that they had at the end of execution of the forward transaction of the associated saga statement. If such variables are changed after the saga statement is executed this will not be seen in the compensation. To support such behaviour it is possible to associate data with a saga through functions that are indexed with the current saga (can be accessed by `current_saga()`).

12 Physical database design

This section describes some AMOSQL commands for database space and performance tuning.

12.1 Indexing

The system supports indexing on any argument or result of stored functions. Indexes can be *unique* or *non-unique*. A unique index prohibits more than one different value of the argument or result. The cardinality constraint `key` of stored functions (See “Cardinality Constraints” on page 17.) is implemented as unique indexes. Thus by default the system puts a unique index on the first argument of stored functions. That index can be made non-unique by suffixing the first argument declaration with the keyword `nonkey` or to specify `bag of` for the result, in which case a non-unique index is used instead.

For example, in the following function there can be only one `name` per `person`:

```
create function name(person)->charstring as stored;
```

By contrast, `names` allow more than one name per `person`:

```
create function names(person p nonkey)->charstring nm as stored;
```

alternatively

```
create function names(person p)->bag of charstring nm as stored;
```

Any argument or result declaration can be suffixed with the keyword `key` to indicate the position of a unique index. For example, the following definition prohibits two persons to have the same name:

```
delete function person.name; /* Remove old name function */

create function name(person p)->charstring nm key as stored;
```

Named non-unique indexes can be created on any arguments or results with the statement:

```
create-index-stmt ::=
  CREATE INDEX index-name ON index-spec-commalist ;
index-spec ::=
  function-name '(' argres-name ')'
```

For example:

```
create index i1 on name(nm), names(charstring);
```

creates two indexes on the result of `name` and of `names`, respectively.

Notice that one may use the name of the type of an argument or result to specify the index position when unique, as in the example.

Named indexes are deleted by

```
delete-index-stmt ::=
  DELETE INDEX index-name ;
```

For example, to delete the two indexes above do:

```
delete index i1;
```

There always has to be at least one index left on each stored function. Thus `delete index` is a dummy operation if one tries to delete the last remaining index.

To save space it is sometimes possible to delete the default index on the first argument of a stored function. For example, suppose we store a table mapping parts to identifiers with an index on the identifier:

```
create type part;
create function partid(part p)->integer id key as stored;
```

`partid` will have two indexes, one on `p` and one on `id`. To drop the index on `p`, do the following:

```
create index dummy on partid(p);
 delete index dummy;
```

12.2 Clustering

Functions can be clustered by creating multiple result stored functions, and then each individual function can be defined as a derived function.

For example, to cluster the attributes `name` and `address` of persons one can define:

```
delete function person.name;
create function personprops(person p) ->
  <charstring name,charstring address> as stored;
create function name(person p) -> charstring nm
  as select nm for each charstring a
        where personprops(p) = <nm,a>;
create function address(person p) -> charstring a
  as select a for each charstring nm
        where personprops(p) = <nm,a>;
```

Clustering does not improve the execution time performance significantly in a main-memory DBMS such as AMOS. However, clustering can decrease the database size considerably.

13 System functions

This section describes the built-in system AMOS functions.

13.1 Comparison operators

The built-in, infix comparison operators are:

```
=(object x, object y) -> boolean   (infix operator =)
!=(object x, object y) -> boolean  (infix operator !=)
>(object x, object y) -> boolean   (infix operator >)
>=(object x,object y) -> boolean   (infix operator >=)
<(object x, object y) -> boolean   (infix operator <)
<=(object x,object y) -> boolean   (infix operator <=)
```

All objects can be compared. Strings are compared by characters, lists by elements, OIDs by identifier numbers. Equality between a bag and another object denotes set membership of that object. The comparison functions can, of course, be overloaded for user defined types.

13.2 Arithmetic functions

```
abs(number x) -> number y
div(number x, number y)   -> number z (infix operator /)
max(object x, object y)   -> object z
min(object x, object y)   -> object z
minus(number x, number y) -> number z (infix operator -)
mod(integer x, integer y) -> integer z
plus(number x, number y)  -> number z (infix operator +)
times(number x, number y) -> number z (infix operator *)
iota(integer l, integer u)-> bag of integer z
sin(number x) -> number z
cos(number x) -> number z
tan(number x) -> number z
ln(number x)  -> number z
sqr(number x) -> number z
```

`iota` constructs bag of integers between `l` and `u`.
For example, to execute `n` times AMOSQL statement `stmt` do:

```
for each integer i where i = iota(1,n)
  stmt;
```

13.3 Aggregation functions

Some of these system functions are described in “Subqueries and Aggregation Operators” on page 21 .

```
count(bag of object o) -> integer c
```

Number of objects in bag `o` (“Subqueries and Aggregation Operators” on page 21).

```
in(bag of object b) -> bag of object o
in(vector v) -> bag of object o
```

Extracts elements of bags and vectors

```
maxagg(bag of object x) -> object y
```

Largest number in bag

```
minagg(bag of object x) -> object y
```

Smallest number in bag.

```
notany(bag of object o) -> boolean b
```

Test if bag empty. Logical NOT EXISTS.

```
some(bag of object x) -> boolean b
```

Test if there are any elements in bag. Logical EXISTS.

```
sum(bag of integer x) -> number s
sum(bag of real x) -> number s
```

Sum uniform bags of numbers.

13.4 Accessing the type system

```
allfunctions() -> bag of function f
allfunctions(type t) -> bag of <integer pos, function f, integer kind>
allfunctions(type t,integer pos, integer kind) -> bag of function f
```

Returns all functions, all functions that take as argument or return a given type, all functions that take or return a type at a given postion, respectively:
`f`: The function.
`pos`: The position number. (1st is 1, etc.)
`kind`: A number indicating if it is an argument (`kind` = 0) or a result (`kind` = 1).

```
allobjects () -> bag of object o
alltypes() -> bag of type t
```

All functions, objects, and types, respectively, in the database.

```
subtypes(type t) -> bag of type s
supertypes(type t) -> bag of type s
```

The types immediately below/above type `t` in the type hierarchy.

```
allsupertypes(type t) -> bag of type s
```

All types above `t` in the type hierarchy.

```
typesof(object o) -> bag of object t
```

The types of an object.

```
allobjects(type tp)-> bag of object t
createobject(type tp) -> object t
```

Get all instances `t` of a given type `tp` or create a new instance `t` of a given type `tp`.

```
functionnamed(charstring nm) -> function fn
kindoffunction(function f) -> charstring knd
name(function fn) -> charstring nm
typenamed(charstring nm) -> type t
```

```
name(type t) -> charstring nm
objectname(object o, charstring nm) -> boolean
usedwhere(function f) -> function c
useswhich(function f) -> function c
```

`functionnamed` returns the function named `nm`.
`kindoffunction` returns the kind of the function `f` as a string. The result can be one of `'stored', 'derived', 'foreign'` or `'overloaded'`. `nameoffunction` returns the name of the function `f`.
`typenamed` returns the type named `nm`.
`nameoftype` returns the name of the type `t`.
`objectname` returns TRUE if the object `o` has the name `nm`.
`usedwhere` returns the functions calling the function `f`.
`useswhich` returns the functions called from the function `f`.

```
resolvents(function g) -> bag of function r
```

The resolvents of an overloaded function `g`.

```
resolventtype(function fn) -> bag of type t
```

The types of the first arguments of the resolvents of function `fn`.

```
argrestypes(function fn nonkey) -> <integer pos,type tp,integer kind>
argrestypes(charstring fname nonkey) -> <integer pos,type tp,integer
kind>
```

Returns for each argument or result of a function:
`pos`: The position number. (1st is 1, etc.)
`type`: The type.
`kind`: A number indicating if it is an argument (`kind` = 0) or a result (`kind` = 1).

```
addtype(usertype tp,usertypeobject o) -> boolean
remtype(usertype tp,unsertypeobject o) -> boolean
```

Procedures to add/remove type `tp` to/from object `o`.

```
cardinality(type t) -> integer c
```

Number of object of type `t`.

13.5 Query optimizer tuning

```
optmethod(charstring new) -> charstring old
```

Three optimization modes for AMOSQL queries can be chosen. The built-in function

```
call optmethod("name");
```

The name of the old optimization method is returned. Changes the optimization method to `name`, which can be one of:
`ranksort`: (default) which is fast but not always optimal.
`exhaustive`: which is optimal but it may slow down the optimizer considerably.
`randomopt`: which is a combination of two heuristics: Iterative improvement and Sequence heuristics [4].
`randomopt` can be tuned by using the function

```
call optlevel(i,j);
```

where `i` and `j` are integers specifying number of iterations in Iterative improvement and sequence heuristics respectively. Default settings is `i`=5 and `j`=5.

```
    reoptimize(charstring fn) -> boolean
```

Reoptimize function named `fn`. if `fn` is equal to the string `"*ALL*"` then all functions are reoptimized.

```
    costhint(charstring fn,charstring bpat,object q)->boolean
```

Declare cost hint `q` for the AMOSQL resolvent function named `fn` and the binding pattern `bpat`. This cost hint feature is explained in **AMOS System Manual** and in [1]. The cost hint can be a vector of two elements, `{cost,fanout}`, in case the cost to execute `fn` is constant. It can also be the name of an AMOSQL function returning the cost and the fanout.

13.6 Temporal support in AMOS

AMOS supports three data types for referencing time. `Timeval` is a type for specifying absolute time points and `time` and `date` are types for relative time points.

Timevals are written as |year-month-day/hour:minute:second|, e.g. `|1995-11-15/12:51:32|`.

Times are written as |hour:minute:second|, e.g. `|12:51:32|`.

Dates are written as |year-month-day|, e.g. `|1995-11-15|`.

The follwing functions exist for timevals, times, and dates.

```
    now() -> timeval
    time() -> time
    date() -> date
    timeval(integer,integer,integer,integer,integer,integer) -> timeval
    time(integer,integer,integer) -> time
    date(integer,integer,integer) -> date
    time(timeval) -> time
    date(timeval) -> date
    date_time_to_timeval(date, time) -> timeval
    year(timeval) -> integer
    month(timeval) -> integer
    day(timeval) -> integer
    hour(timeval) -> integer
    minute(timeval) -> integer
    second(timeval) -> integer
    year(date) -> integer
    month(date) -> integer
    day(date) -> integer
    hour(time) -> integer
    minute(time) -> integer
    second(time) -> integer
    timespan(timeval, timeval) -> <time, integer usec>
```

13.7 Miscellaneous functions

```
    cd(charstring dir) -> charstring r
    eval(charstring stmt) -> object r
    pwd() -> charstring dir
    quit() -> boolean
    stop() -> boolean
    system(charstring cmd) -> boolean
```

`cd` changes the current working Unix directory to `dir` and returns its full name.
`eval` parses and evaluates the AMOSQL statement `stmt`. Currently the result must be single-valued, otherwise

only the first element of the result tuple is returned.
`pwd` returns the full name of the current Unix working directory.
`quit` quits AMOS.
`stop` exits the AMOS top loop and returns to the program that called it. Useful when calling AMOS from other systems.
`system` executes the Unix command `cmd`. No value returned.

14 Recovery–Storage System

This section describes the recovery system[Kar94] in AMOS and how to configure this from within the AMOS system using AMOSQL.The recovery system in AMOS is responsible for the automatic persistency of the transactions on the database image.

The recovery system is activated by simply typing:

```
recovery on;
```

and deactivated using

```
recovery off;
```

14.1 Configure

The recovery system uses the ping-pong method for saving images, so images are written to two alternating places at a time interval. This interval, in minutes, can be specified using:

```
recovery interval 30;
```

At each commit the log-information is flushed onto the disk, when an excessive amount of information is changed in a short period the log-file grows fastly. In order to limit the size of this file an image can be forced to be saved when the log-file reaches a specified size. This size is specified in kilobytes (KB):

```
recovery maxlog 128;
```

The current state of the recovery system is shown by just typing recovery:

```
recovery;
  Persistency system is active.
  Interval for saving image is 30 minutes.
  Maximum size for the log is around 128 KB.
```

These parameter can be set and be changed together, the syntax is:

```
recovery (ON | OFF) [interval <minutes>] [maxlog <kb>];
```

Before shutting down the AMOS system an image should be saved this should be done using:

```
recovery quit;
```

14.2 Recovery

If the AMOS system was incorrectly terminated, a recovery action is taking place at startup time. This is automatic for an image that had recovery activated. When starting AMOS the image-file “amos.dmp” is loaded by default, this is actually a symbolic link to the last fully saved image and “amos.log.first” and “amos.log.second” exists only if the system was incorrectly exited. So just by starting amos at the unix prompt recovers:

```
unix> amos
NIL
AMOS 0.1a, (c) CAELAB 1993
```

15 Miscellaneous

The transaction logging can be turned on and off with the `logging` statement:

```
logging-stmt ::= LOGGING toggle

toggle ::= (ON | OFF)
```

The AMOSQL statement `rollback` does nothing when logging is turned off. It is often practical to turn off logging when building large databases, since the system then consumes much less space.

The system will print the execution time of each top level AMOSQL statement by issuing the `timing` statement:

```
timing-stmt ::= TIMING toggle
```

The result printing of the results of AMOSQL statements can be toggled with the `echo` statement:

```
echo-stmt ::= ECHO toggle
```

The image size can be increased arbitrarily with the `imagesize` statement:

```
imagesize-stmt ::= IMAGESIZE integer-constant
```

The system automatically increases the image size with 25% when the image is full. On some systems it is faster to increase the image size to the expected final database size before building a large database.

To redirect the AMOSQL input from a file use:

```
redirect-stmt ::= ’<’ string-constant
```

For example

```
< ’person.AMOSQL’;
```

16 Bugs

If You should find any bugs in a released AMOS, then send a mail to `amos-bugs@ida.liu.se` describing the bug. If possible include the an example detailed enough to recreate the bug. If AMOS dumps core You could run `gdb` or some other debugger and look at the stack to see in what function the fault occured. Using `gdb` You would type the following

```
gdb amos core
>where
```

and include the stack trace in the email.

References

[1] W.Litwin, T.Risch: Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[2] D.Fishman et al.: "Overview of the IRIS DBMS", in W.Kim, F.H.Lochovsky (eds.): *Object-Oriented Concepts, Databases, and Applications*, ACM Press, Addison-Wesley, 1989.

[3] Y.E.Ioannidis, Y.C.Kang: Randomized Algorithms for Optimizing large join queries, *Proc. ACM SIGMOD Conf.*, Atlantic City, 1990, pp 312-321.

[4] J.Näs: Randomized optimization of object oriented queries in a main memory database management system, *Master's thesis*, *LiTH-IDA-Ex 9325* Linköping University 1993.

[5] P.G.Selinger et al: Access Path Selection in a Relational Database Management System, *Proc. ACM SIGMOD Conf.*, Boston, 1979, pp 23-34.

[6] J.D.Ullman: Principles of Database and Knowledge-Base Systems, Volume I and II, *Computer Science Press*, 1988 and 1989.

[7] M.Werner: A Client-Server Interface for AMOS, *CAELAB Memo*, Linköping University 1994.

[8] J.S.Karlsson: An Implementation of Transaction Logging and Recovery in a Main Memory Resident Database System, *Masters's thesis, LiTH-IDA-Ex-9404* Linköping University 1994.

[9] S. Flodin: An Incremental Query Compiler with Resolution of Late Binding, *Research Report LiTH-IDA-R-94-46*, Linköping University 1994

[10] S. Flodin, T. Risch, Processing Object-Oriented Queries with Invertible Late Bound Functions, *Proc. of 1995 VLDB conference*

[11] M. Sköld, Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, *Lic Thecis No 452,* Linköping University.

[12] M. Sköld, E.Falkenroth, T.Risch, Rule Contexts in Active Databases - A Mechanism for Dynamic Rule Grouping, *In the RIDS'95 (Rules in Database Systems)*, Athens, Greece, September 25-27, 1995, Springer Lecture Notes in Computer Science, pp. 119-130, ISBN 3-540-60365-4