Implementation of the Evolving Algebra System[1]

Dag Diesen
Department of Informatics
University of Oslo

March 1995

[1]This report is Appendix B to the report: Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages.

Contents

1 Evolving Algebra **1**

1.1 Definition of the Evolving Algebra 1

1.2 A Simple Example 3

1.2.1 Signatures, Initial Values and Transition Rules 3

2 Interpreting the Evolving Algebra **5**

2.1 The Simple Stack Example 5

2.1.1 Signatures 5

2.1.2 Commands 6

2.1.3 Initial Values 7

2.1.4 The Transitions 7

2.2 The Levels of the Interpreter 8

2.3 The Fixed Part of Evolving Algebra Definitions 9

2.3.1 Definition of an Algebra List 9

2.3.2 The Signature 9

2.3.3 The Initial Function Assignment 10

2.3.4 The Initial Set Assignment 11

2.4 The Definition of the Transition 11

2.4.1 The Predicate 12

2.5 The Update Definitions Level 13

2.5.1 Definition of an Function Update 13

2.5.2 Definition of Function Procedure Update 14

2.5.3 Definition of Contraction 14

2.5.4 Definition of the Universe Update 14

2.6 The Universe Extension Level 15

2.7 The Universe Function Update Level 16

2.8 Some other BNF-Definitions 16

2.9 Interpreter Commands 17

2.9.1 Bind Scheme Symbols to a Function Name 17

2.9.2 Bind Scheme Symbols to a universe Name 18

2.9.3 Execute the Transitions 18

2.9.4 The Reset Command 19

2.9.5 Print the Statistics 19

2.9.6 Read from File 19

2.9.7 Load Scheme Definitions 20

2.9.8 Execute a Scheme Expression 20

2.9.9 Display User Bindings 21

2.9.10 Fetch Data Assigned to the Function Name 21
2.9.11 Fetch Data Assigned to the Universe Name 21
2.9.12 Load System Procedures 21
2.10 Update of statistical records 22

3 Processing 23
3.1 The Internal Data Structure 23
3.1.1 List of Function Definitions 24
3.1.2 Universe Definitions 24
3.1.3 List of Algebra Definitions 25
3.1.4 List of Signature Definition 25
3.1.5 List of Transitions 26
3.1.6 A Transition 26
3.1.7 An update 26
3.1.8 A Universe Extension 26
3.1.9 A Universe function update 28
3.1.10 Statistical data 28
3.2 Processing of the Initial Values 29
3.3 Processing of the Transitions 30
3.3.1 Processing of the Updates 31
3.4 Represent the Functions 32
3.4.1 Evolving Algebra Environment Procedures 33
3.4.2 The Evolving Algebra Environment 33
3.5 Some Written Evolving Algebra Environment Procedures . . 36
3.5.1 Some Lookup procedures 37
3.5.2 The Assign Procedures 38
3.5.3 A Universe Extension Procedure 41

4 Output and Statistics 42
4.1 Result of the Computation 42
4.1.1 Definitions and Commands 42
4.1.2 Executing the Transitions 42
4.2 Statistics 42
4.2.1 Definition Statistic for an Evolving Algebra Specification 42
4.2.2 Execution Statistic for an Evolving Algebra Specification 45
4.2.3 Example of the statistic 50
4.3 Error Handling 50
4.3.1 Input Errors 50
4.3.2 Errors when Executing a Scheme Definitions 50

A Execution of the Stack Example 52

B Output when Parsing the Stack Example 55

C Defintition Statistic of the Stack Example 57

D Numbers Assigned to Transitions and Updates 59

E Execution Statisitic of the Stack Example **61**

List of Figures

2.1 The interpreter levels 8

3.1 The global lists 23
3.2 List of definitions 25
3.3 Transition data structure 27
3.4 Elements in a universe update object 29
3.5 Top level statistical data structure 29
3.6 Statistical data records 30
3.7 The function values are stored in a table 39
3.8 The function values are computed by a procedure 40
3.9 Function values computed by a procedure or stored in a table 40

Chapter 1

Evolving Algebra

1.1 Definition of the Evolving Algebra

In order to give background to the implementation of evolving algebra (also known as dynamic algebra) we need the definition of evolving algebra.

The definitions of evolving algebra are taken from [BR91c] and from [Bör90a].

Definition 1 (Evolving algebra [BR91c]) *A evolving algebra of a given signature is a pair (A, T) consisting of a finite many sorted partial first order algebra A and a finite set T of transition rules of the signature.*

Definition 2 (A transition rule [BR91c]) *A transition rule is an expression of the form*

> ***If*** *condition*
> ***then***
> *update*$_1$
> ⋮
> *update*$_k$

where condition is a boolean expression. If this expression is evaluated to "true" the updates belonging to the transition rule is executed.

There is to kinds of updates, the function update and the universe update.

To express redefinition of a function at one point we define the function update expression.

Definition 3 (Function update [BR91c]) *A function update is an expression of the form*

$$f(t_1, \ldots t_n) := t$$

If we need to add new elements to one of the universes and use the new elements in one or more function updates, we define the universe update expression.

Definition 4 (Universe update (simple) [Bör90a]) *A universe update is an expression of the form*

$$
\begin{array}{ll}
\textit{EXTEND} & U \;\; \textit{by temp} \\
\textit{WITH} & F_1 \\
 & \vdots \\
 & F_k \\
\textit{ENDEXTEND} &
\end{array}
$$

where U is the universe to be extended, temp is the constant holding the new element to be added to the universe and expressions $F_1, \ldots, F_k$ are function updates used as part of the universe update. The name temp *may occur in the function updates.*

A far more complicated way to define universe update is taken from [BR91c].

Definition 5 (Universe updates (complicated) [BR91c]) *The universe updates may be written as an expression of the form*

$$
\begin{array}{ll}
\textit{EXTEND} & D_1 \;\; \textit{by } temp(1,1), \ldots, temp(1, t(1)) \\
 & \vdots \\
\textit{EXTEND} & D_l \;\; \textit{by } temp(l,1), \ldots, temp(l, t(l)) \\
\textit{WITH} & F_1 \\
 & \vdots \\
 & F_k \\
\textit{ENDEXTEND} &
\end{array}
$$

where $D_1, \ldots, D_l$ are universes, $t(1), \ldots, t(l)$ are terms, and $F_1, \ldots, F_k$ are function updates.

The names $temp(i, j)$ with parameters i, j may occur in the function updates.

The terms $t(1), \ldots, t(l)$ are evaluated to $n_1, \ldots, n_l$, where n_i is the number of elements to be added to the universe D_i for each $1 \leq i \leq l$.

The element $temp(i, j)$ to be added to D_i is created for each $1 \leq i \leq l, 1 \leq j \leq n_i$.

All function updates $F_1, \ldots, F_k$ are simultaneously executed for each $1 \leq i \leq l, 1 \leq j \leq n_i$.

The definition of the universe update above describes the basis for how the universe update is implemented in the evolving algebra interpreter. In the case that more than one element are to be added to the universe, it is possible to refer to whichever of the new elements in the function update statements within the universe update.

In addition we have implemented the possibility to create instances of the function updates for all elements added to the universe(s) in the universe extension. Each instance of a function update will refer to a unique combinations of new elements referred in the function update.

So we can give the function update:

$$a(temp(i, j)) := temp(i, j + 1)$$

meaning that the $a(temp(i, j))$ points to its successor element $temp(i, j + 1)$ for all elements j added to the universe i, where the definition above gives

some meaning. If n elements is added to the universe i then $n-1$ instances will be created of the function update above (no meaning can be given to the instance where the new element n is referred on the left hand side, so the n'th instance of the function update are to be discarded).

1.2 A Simple Example

I will here describe a very simple language consisting of the following three commands:

- *Pop* : Pop an element from the stack.
- *Push* : Push an element to the stack.
- *Stop* : Halt the execution of the program.

In addition the program will halt if the stack is empty.

1.2.1 Signatures, Initial Values and Transition Rules

The following signatures will be used:

$stack$:	$STACK$
$value$:	$VALUE$
pop:	$STACK \rightarrow STACK$
top:	$STACK \rightarrow VALUE$
$push$:	$(STACK \times STACKEL) \rightarrow STACK$
$emptystack$:	$STACK \rightarrow BOOL$
$cmds$:	$CMDS$
$next$:	$CMDS \rightarrow CMDS$
$first$:	$CMDS \rightarrow COMMAND$
$halt$:	$STATE$

Assume the following initial values given:

$$halt = 0$$
$$cmds = (Push, Push, Pop, Stop)$$
$$stack = (Temp)$$

The following four transition rules will describe the semantics of the programming language to the desired abstraction level.

The first transition rule gives the semantic for the *Pop* command.

if $halt \neq 1$
    & $first(cmds) = Pop$
then
    $value := top(stack)$
    $stack := pop(stack)$
    $cmds := next(cmds)$

The second transition rule gives the semantic for the *Push* command.

$halt \neq 1$
& $first(cmds) = Push$
then
extend STACKEL by $temp$
$stack := push(stack, temp)$
endext
$cmds := next(cmds)$

The third transition rule gives the semantic for the $Stop$ command.

if $halt \neq 1$
& $first(cmds) = Stop$
then
$halt := 1$

The last transition rule gives the semantic for what to do when the stack is empty.

if $halt \neq 1$
& $empty(stack)$
then
$halt := 1$

Chapter 2

Interpreting the Evolving Algebra

This chapter describe parsing of evolving algebra definitions and commands.

2.1 The Simple Stack Example

The simple example of stack operations presented in section 1.2 is given again as a complete example of input to the interpreter.

2.1.1 Signatures

The signatures of the functions are given below. The interpreter use the signatures when making the internal datastructure of the Evolving Algebra definitions.

```
%reset before loading
reset
%         STACK OPERATION

%               SIGNATURE

signature value : VALUE
signature stack : STACK
signature pop : (STACK --> STACK)
signature top : (STACK --> VALUE)
signature push : ((STACK x STACKEL) --> STACK)
signature emptystack : (STACK --> BOOL)

signature cmds : CMDS
signature next : (CMDS --> CMDS)
signature firstel : (CMDS --> COMMAND)

signature halt : STATE
```

2.1.2 Commands

The interpreter needs definitions of how the functions should act and how updates are to be performed. Special Lisp procedures are loaded and bound to the function and universe symbols in order to tailor this needs.

In this example the constants are bound to some standard Scheme procedures performing lookups and function updates. The functions which executes the operation on the stack and list of stack commands are bound to Scheme procedures performing operations on lists.

The universe names are bound to a procedure which add new elements to a datastructure representing a finite set. Here a list structure is used to represent the finite sets.

```
%                 COMMANDS
% Load standard user environment procedures from file!
%
loadproc "Ea-system-lib/ea-std-user-extension.scm";
loadproc "Ea-system-lib/ea-std-user-update.scm";
loadproc "Ea-system-lib/ea-std-user-lookup.scm";

% Load procedures maintaining lists
loadproc "Ea-system-lib/ea-std-user-list.scm";

% Assignments
% Assign procedure to the function  or extension symbol

% assign <function-symbol>, <lookup-procedure>,
%                 <update-procedure>, <function-message-symbol>;
assignfunc stack, constant-std-lookup-data, user-update-constant,
           std-const-dta;
assignfunc cmds, constant-std-lookup-data, user-update-constant,
           std-const-dta;
assignfunc halt, constant-std-lookup-data, user-update-constant,
           std-const-dta;
assignfunc value, constant-std-lookup-data, user-update-constant,
           std-const-dta;
assignfunc pop, tail-from-list, dummy-func, upd-not-perm;
assignfunc push, add-to-list, dummy-func, upd-not-perm;
assignfunc top, first-from-list, dummy-func, upd-not-perm;
assignfunc emptystack, empty-list, dummy-func, upd-not-perm;
%
assignfunc next, tail-from-list, dummy-func, upd-not-perm;
assignfunc firstel, first-from-list, dummy-func, upd-not-perm;
%
% assign <universe-extension-procedure>, <universe-message-symb>;
assignuniverse STACK, std-ext-collection, newsymbol;
assignuniverse STACKEL, std-ext-collection, newsymbol;
assignuniverse CMDS, std-ext-collection, newsymbol;
assignuniverse VALUE, std-ext-collection, newsymbol;
```

```
assignuniverse BOOL, std-ext-collection, newsymbol;
assignuniverse COMMAND, std-ext-collection, newsymbol;
assignuniverse STATE, std-ext-collection, newsymbol;
```

2.1.3 Initial Values

The initial values are given to function and universe symbols before the transitions are executed. Here the command sequnence of stack operation is given, the value of the stack is set to nonempty and the state variable *halt* is set to zero, which means do not halt. We do not explicit initialize the contents of the universes here, although it can be done.

```
%                INITIAL-VALUES

% Assignments are performed in sequence.

initial halt :== 0
initial cmds :== [("Push" "Push" "Pop" "Stop")]
initial stack :== [("Temp")]
```

2.1.4 The Transitions

The transitions performs the stack operations *Pop* and *Push* and halts if the *Stop* command is given or if the stack becomes empty.

```
%                THE TRANSITIONS
% The pop command

if ( = (halt,0) &
     (! emptystack(stack)) &
     =  (firstel(cmds) , "Pop") );
   funcupdate value := top (stack) ;
   funcupdate stack := pop (stack) ;
   funcupdate cmds := next (cmds) ;
endupdates

% The push command

if ( = (halt,0) &
     = (firstel(cmds) , "Push") );
   extend
      extenduniverse STACKEL;
   withupdates
      funcupdate stack := push(stack,temp(STACKEL,1));
   endextend
   funcupdate cmds := next (cmds) ;
endupdates

if ( = (halt,0) &
```

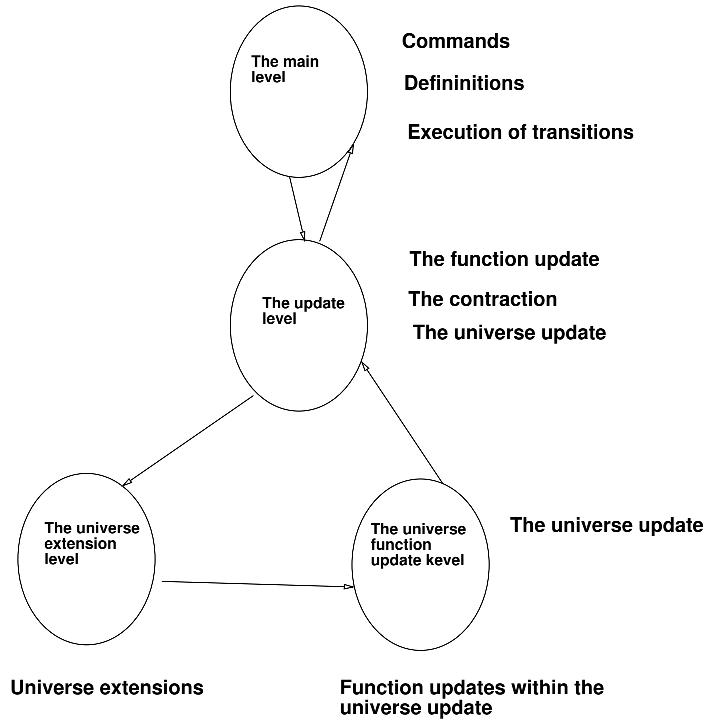

Figure 2.1: The interpreter levels

```
    = (firstel(cmds) , "Stop")) ;
  funcupdate   halt := 1 ;
endupdates

if ( = (halt,0) &
    emptystack(stack) );
  funcupdate   halt := 1 ;
endupdates
```

2.2 The Levels of the Interpreter

At the main level the interpreter displays a prompt and expects the user to type some input. The input may be a *command* or some type of an *evolving algebra definition*.

The evolving algebra interpreter will parse the given expression, update the internal data structure or execute the command, print the result of the parsing and give you the next prompt.

The interpreter will enter the update level, when a definition of a transition is to be parsed. If a universe extension is given, the interpreter will first enter the universe extension level and then enter the universe function update level. At the universe extensions level all universe extensions within the universe update will be parsed. All universe function updates will then be parsed at the universe update level. See Figure 2.1 for an overview.

The execution of the transitions may be invoked from the main level when all definitions is given.

2.3 The Fixed Part of Evolving Algebra Definitions

2.3.1 Definition of an Algebra List

If the user types “algebra” as the first symbol, then a definition of a named algebra is to be parsed. The definition consists of a list of sets which is part of the algebra and a list of lists of name of similar functions.

The algebra list is saved as part of the internal data structure, but not used in the system at present. Therefore this definition may be omitted

Syntax of the Algebra Definition

The syntax of the algebra definition is given in the following BNF-expressions:

⟨Algebra-definition⟩	::=	`algebra` ⟨name⟩ : ⟨algebra-def-list⟩
⟨Name⟩	::=	⟨Lower-case-letters⟩
⟨Algebra-def-list⟩	::=	(⟨sets⟩ ⟨list-of-similar-functions⟩)
⟨Sets⟩	::=	⟨Set⟩ **|** ⟨Sets⟩ , ⟨Set⟩
⟨List-of-similar-functions⟩	::=	; ⟨Functions⟩ **|** ⟨List-of-similar-functions⟩ ; ⟨Functions⟩
⟨Functions⟩	::=	⟨Function-name⟩ **|** ⟨Functions⟩ , ⟨Function-name⟩
⟨Function-name⟩	::=	⟨Lower-case-letters⟩

The syntax of ⟨Set⟩ will be defined in section 2.3.2.

Example of an Algebra Definition

An algebra definition may look like:

```
algebra stack : (LOCATION,VALUE,{0,1};top,bottom;stop;add,subtract)
```

2.3.2 The Signature

If “signature” is the first symbol, then a signature definition will be parsed. A signature definition consists of a list of function names and a set expression. The set expression denote the type of the function.

The signature is saved in the internal data structure. In addition all universe names and function names are saved in the internal data structure to be used later.

Syntax of the Signature Definition

The syntax of the signature definition is given in the following BNF-expressions:

```
⟨Signature⟩          ::= signature ⟨Functions⟩ :
                         ⟨Type-struct-structure⟩
⟨Type-struct⟩        ::= ⟨sets⟩ | ( ⟨type-struct-expr⟩ )
⟨Type-struct-expr⟩   ::= ⟨Seq-type⟩ | ⟨Cross-product⟩ |
                         ⟨Function-type⟩
⟨Seq-type⟩           ::= ⟨Type-struct⟩ *
⟨Cross-product⟩      ::= ⟨Type-struct⟩ x ⟨Type-struct⟩ |
                         ⟨Cross-product⟩ x ⟨Type-struct⟩
⟨Function-type⟩      ::= ⟨Type-struct⟩ --> ⟨Type-struct⟩ |
                         ⟨Function-type⟩ --> ⟨Type-struct⟩
```

The syntax of a set definitions is as follows:

```
⟨Set⟩                    ::= ⟨Collection⟩ | ⟨Universe-name⟩ |
                             [ ⟨Set-expression⟩ ]
⟨Set-expression⟩         ::= ⟨Union-of-sets⟩ | ⟨Difference-of-sets⟩ |
                             ⟨Intersection-of-sets⟩
⟨Union-of-sets⟩          ::= ⟨Set-expr-part⟩ + ⟨Set-expr-part⟩ |
                             ⟨Union-of-sets⟩ + ⟨Set-expr-part⟩
⟨Difference-of-sets⟩     ::= ⟨Set-expr-part⟩ - ⟨Set-expr-part⟩ |
                             ⟨Difference-of-sets⟩ - ⟨Set-expr-part⟩
⟨Intersection-of-sets⟩   ::= ⟨Set-expr-part⟩ & ⟨Set-expr-part⟩ |
                             ⟨Intersection-of-sets⟩ & ⟨Set-expr-part⟩
⟨Set-expr-part⟩          ::= ⟨Collection⟩ | ⟨Universe-name⟩ |
                             [ ⟨Set-expression⟩ ] |
                             ( ⟨Set-expression⟩ )
⟨Collection⟩             ::= { ⟨Elements⟩ }
⟨Elements⟩               ::= ⟨Element⟩ | ⟨Elements⟩ , ⟨Element⟩
⟨Element⟩                ::= ⟨Value⟩
⟨Universe-name⟩          ::= ⟨Upper-case-letters⟩
```

Example of Signature Definitions

```
signature stop : {0,1}
signature userinput : USERINPUT
signature write : (( [ VALUE + TEXT ] ) --> WRSTATUS
signature cross-product : (A x B)
```

2.3.3 The Initial Function Assignment

If the symbol “initial” is typed as the first symbol, then an initial assignment of a function value to a function expression is parsed.

The function value from the right hand side expression will be computed. Then the value will be assigned as an initial value to the function expression

on the left hand side. If arguments are given in the left hand side function expression the values of the arguments are used as a key to the function value space.

Syntax of the Initial Function Assignment Definition

The syntax is given in the BNF-expressions below:

⟨Initial-value-def⟩ ::= `initial` ⟨Assign-to-func⟩ `:==` ⟨Init-func-value⟩ `;`
⟨Assign-to-func⟩ ::= ⟨Func-expression⟩
⟨Init-func-value⟩ ::= ⟨Func-expression⟩ **|** ⟨Value⟩

The syntax of ⟨Func-expression⟩ will be given in 2.4.1.

Examples of Initial Function Assignment Definitions

```
initial read(1) :== 2;
initial stop :== rvalue(1);
```

2.3.4 The Initial Set Assignment

If "initialset" is the first symbol, then an initial assignment of the universe value to a universe name is to be parsed.

A set elements (in form of a list) is given as an initial value to the universe.

Syntax of the Initial Set Assignment Definition

The syntax is given in the BNF-expressions below:

⟨Initial-set-value-def⟩ ::= `initialset` ⟨Assign-to-universe⟩ `:==` ⟨Init-set-value⟩ `;`
⟨Assign-to-universe⟩ ::= ⟨Universe-name⟩
⟨Init-set-value⟩ ::= ⟨Collection⟩

Example of an Initial Set Assignment Definition

```
initialset LOCATION :== {1,2};
```

2.4 The Definition of the Transition

If "if" is given as the first symbol, then a transition is to be parsed.

The user is first expected to give the predicate expression, which is part of the transition. If no error occurs when parsing the predicate expression, the system will enter the update level. All updates belonging to the transition have to be given in the update level.

2.4.1 The Predicate

Syntax of the Update Expression

The syntax of an update expression is given in the BNF-expression below:

```
⟨Update-def⟩ ::= ⟨Condition⟩ ; ⟨Updates⟩ endupdates
⟨Updates⟩    ::= ⟨Update⟩ | ⟨Updates⟩ ⟨Update ⟩
```

The syntax of a condition expression:

```
⟨Condition⟩                ::= ⟨Complex-pred-expr⟩
⟨Complex-pred-expr⟩        ::= ⟨Simple-pred-expr⟩ | (
                               ⟨Complex-pred-expr-part⟩ )
⟨Complex-pred-expr-part⟩   ::= ⟨Or-expr⟩ | ⟨And-expr⟩ |
                               ⟨Not-expr⟩
⟨Or-expr⟩                  ::= ⟨Complex-pred-expr⟩ |
                               ⟨Complex-pred-expr⟩ |
                               ⟨Or-expr⟩ |
                               ⟨Complex-pred-expr⟩
⟨And-expr⟩                 ::= ⟨Complex-pred-expr⟩ &
                               ⟨Complex-pred-expr⟩ |
                               ⟨And-expr⟩ &
                               ⟨Complex-pred-expr⟩
⟨Not-expr⟩                 ::= ! ⟨Complex-pred-expr⟩
⟨Simple-pred-expr⟩         ::= ⟨Pred-form⟩ | ⟨Rel-expr⟩
⟨Rel-expr⟩                 ::= ⟨Rel-operator⟩ ( ⟨Argument⟩ ,
                               ⟨Argument⟩ )
⟨Rel-operator⟩             ::= =/= | = | < | > | <= | >=
⟨Pred-form⟩                ::= ⟨Func-expression⟩
```

Note the difference between | and |. The first character is the "or" in the BNF-expression. The last character is the logical "or" in the predicate expression.

The syntax of a function expression is:

```
⟨Func-expression⟩ ::= ⟨Function-name⟩ |
                      ⟨Function-name⟩ ( ⟨Arguments⟩ )
⟨Arguments⟩       ::= ⟨Argument-expr⟩ |
                      ⟨Arguments⟩ , ⟨Argument-expr⟩
⟨Argument-expr⟩   ::= ⟨Func-expression⟩ | ⟨Value⟩
```

Examples of a Transition

This example simply illustrate the syntax. Do not try to interpret any meaning from this example.

```
if ( = (stop, 0) &
     = (userinput, Stop) ) ;
```

```
%---------- Enter the update level ----------
   funcupdate stopval := true ;
   dispose LESS # f(value) ;
   extend
%---------- Enter the universe extension level ---------
      extenduniverse LOCATION # f(5) ;
      extenduniverse TEMP ;
      withupdates
%---------- Enter the universe function update level -----
      funcupdate read(temp(LOCATION,3)) := true ;
      endextend
%---------- Return to the update level ---------
  funcupdate true := false
  endupdates
%---------- Return to the main level -----------
if stopval;
   funcupdate stop := 1;
   endupdates
```

2.5 The Update Definitions Level

The update level displays a prompt. The user may type an update expression or quit the update level by typing "endupdates".

Three types of an update expression are allowed. The three types may be function update, universe update and contraction.

Syntax of an Update Definition

⟨Update⟩ ::= ⟨Function-update⟩ | ⟨Universe-update⟩ |
⟨Contraction⟩

2.5.1 Definition of an Function Update

If the user types "funcupdate", then a definition of a function update will be parsed.

The function update definition is saved in the internal data structure.

Syntax of the Function Update Definition

⟨Function-Update⟩ ::= `funcupdate` ⟨Redef-function⟩ `:=` ⟨Redef-value⟩
⟨Redef-function⟩ ::= ⟨Func-expression⟩
⟨Redef-value⟩ ::= ⟨Argument-expr⟩

Example of an Function Update Definition

```
funcupdate top := subtract(top);
```

2.5.2 Definition of Function Procedure Update

If the user types "funcreplace", then a definition of a function procedure update will be parsed.

The function procedure update expression consists of a function name on the left hand side of the delimiter and a function name on the right hand side. The procedure computing the function values for function named on the left hand side is to be replaced by the similar procedure connected to the function named on right hand side.

The function procedure update definition is saved in the internal data structure.

Syntax of the Function Procedure Update Definition

⟨Procedure-update⟩ ::= `funcreplace` ⟨Function-name⟩ `:=`
⟨Function-name⟩

Example of an Function Procedure Update Definition

```
funcreplace old := new;
```

2.5.3 Definition of Contraction

If the user types "dispose", then a definition of a contraction is to be parsed.

The definition of the contraction will be saved in the internal data structure.

At present time the definition of the contraction will not be used in the system.

Syntax of the Contraction Definition

⟨Contraction⟩ ::= `dispose` ⟨Universe-name⟩ `#`
⟨Dispose-value⟩
⟨Dispose-value⟩ ::= ⟨Func-expression⟩

Example of a Contraction Definition

```
dispose LESS # f(value);
```

2.5.4 Definition of the Universe Update

If "extend" is the first symbol, then the system will enter the universe extension level.

Syntax of the Universe Update Definition

⟨Function-universe-update⟩ ::= `extend` ⟨Universe-extensions⟩
`withupdates`
⟨Function-universe-updates⟩
`endextend`

```
⟨Universe-extensions⟩        ::= ⟨Universe-extension⟩ |
                                 ⟨Universe-extensions⟩
                                 ⟨Universe-extension⟩
⟨Function-universe-updates⟩ ::= ⟨Function-universe-update⟩ |
                                 ⟨Function-universe-updates⟩
                                 ⟨Function-universe-update⟩
```

Example of a universe Update Definition

```
    extend
%------- Enters extensions level------
       extenduniverse ALPHA;
       extenduniverse BETA # 2;
       extenduniverse GAMMA # 5 gconst;
       withupdates
%------- Enters universe function updates level---
       funcupdate top := temp(GAMMA,Every(gconst))
       endextend
%------- Return to the updates level -----
```

2.6 The Universe Extension Level

The user may give as many universe extensions expression she want. All universe extensions begins with "extenduniverse" and the universe name. The user may append a number expression or a number expression followed by a constant symbol.

When finished the user types "withupdates" to enter the universe function update level.

Syntax of the Universe Extension Definition

```
⟨Universe-extension⟩ ::= extenduniverse ⟨Universe-name⟩ |
                         extenduniverse ⟨Universe-name⟩
                         ⟨Number-of-objects⟩ |
                         extenduniverse  ⟨Universe-name⟩
                         ⟨Number-of-objects⟩
                         ⟨Gen-constant⟩   ;
⟨Number-of-objects⟩ ::=  # ⟨Number-value⟩
⟨Number-value⟩      ::= ⟨Func-expression⟩ | ⟨Positive-integer⟩
⟨Gen-constant⟩      ::= ⟨Function-name⟩
```

Example of Universe Definitions

```
extenduniverse LOCATION # f(5)
extenduniverse BETA # 8 gc
extenduniverse GAMMA
```

2.7 The Universe Function Update Level

The user may give all function update expressions belonging to the universe update in this level. The given temporary constant may occur on the left hand side of the function update expression.

When finished the user types “endextend” and returns to the update level.

Syntax of the Universe Function Update Definition

```
⟨Function-universe-update⟩ ::= funcupdate
                               ⟨Redef-univ-function⟩ :=
                               ⟨Redef-univ-value⟩
⟨Redef-function⟩           ::= ⟨Univ-func-expression⟩
⟨Redef-value⟩              ::= ⟨Argument-univ-expr⟩
```

The syntax of the universe function expression is as follows:

```
⟨Univ-func-expression⟩ ::= ⟨Function-name⟩ |
                           ⟨Function-name⟩ (
                           ⟨Univ-arguments⟩ )
⟨Univ-arguments⟩       ::= ⟨Argument-univ-expr⟩ |
                           ⟨Univ-arguments⟩ ,
                           ⟨Argument-univ-expr⟩
⟨Argument-univ-expr⟩   ::= ⟨Func-expression⟩ | ⟨Value⟩ |
                           ⟨Pick-new-elem-expr⟩
⟨Pick-new-element-expr⟩ ::= temp ( ⟨Universe-name⟩ ,
                           ⟨Locate-new-element-expr⟩ )
⟨Locate-new-element-expr⟩ ::= ⟨Locate-one⟩ |
                           ⟨Locate-every-inst⟩
⟨Locate-one⟩           ::= ⟨Number-value⟩
⟨Locate-every-inst⟩    ::= Every ( ⟨Number-value⟩ )
⟨Number-value⟩         ::= ⟨Func-expression⟩ |
                           ⟨Positive-integer⟩
```

Example of Universe Function Update Definitions

```
funcupdate top := temp(LOC,3);
funcupdate top-inst(temp(U,Every(i))) :=
           top-inst(temp(U,Every(add(1,i))))
```

2.8 Some other BNF-Definitions

In order to complete the BNF-definitions we will define the following objects:

```
⟨Value⟩        ::= ⟨Value-object⟩
⟨Value-object⟩ ::= ⟨Value-name⟩ | <⟨String1⟩> | "⟨String2⟩" |
                   ⟨Integer⟩ | [⟨Scheme-expression⟩]
```

We give the syntax of some of the objects in an informal way:

- ⟨Comment⟩ If the first non white space character on a line is `%`, then the rest of the line is treated as a comment line.
- ⟨Value-name⟩ is a sequence of letters, where the first letter is an upper case letter and the rest of the sequence consists of lower case letters.
- ⟨String1⟩ is a sequence of characters not containing `>`.
- ⟨String2⟩ is a sequence and characters not containing `"`.
- ⟨Integer⟩ is an integer.
- ⟨Scheme-expression⟩ is a valid Scheme expression.

We refrain from making any formal definitions of the syntax of ⟨Letters⟩, ⟨Upper-case-letters⟩, ⟨Lower-case-letters⟩ and ⟨Positive-integers⟩. The syntax of those objects are obvious.

2.9 Interpreter Commands

This section will give the syntax of some useful commands which may be given to the interpreter at the main level. Some of the commands is necessary to issue in order to run the simulation of the evolving algebra specifications, other commands may be useful in other ways. However, none of the commands below are part of a formal evolving algebra specification.

2.9.1 Bind Scheme Symbols to a Function Name

If "assignfunc" is the first symbol, then an assign command which assigns data to a function symbol will be parsed.

The following data will be assigned to the function name:

- Symbol pointing to a lookup procedure. The procedure will compute the function value.
- Symbol pointing to an update procedure. The procedure will perform the assignment of a function value.
- Symbol which may be used to give a message to one of the procedures.

Syntax of the Function Binding Command

The syntax of the command is given in the following BNF-expressions:

⟨Init-func-assign⟩ ::= `Assignfunc` ⟨Function-name⟩ ,
⟨F-lookup-proc⟩ , ⟨Upd-proc⟩ ,
⟨F-mess-symb⟩ ;

The ⟨F-lookup-proc⟩, ⟨Upd-proc⟩ and ⟨F-mess-symb⟩ are defined as Scheme symbols restricted to contain the alphabetic symbols and in addition the symbols `- # : ? !` . No formal definition of the syntax will be given for those symbols.

Example of a Function Binding Command

```
assignfunc function, lookup, update, func-message;
```

2.9.2 Bind Scheme Symbols to a universe Name

If "assignuniverse" is given, then an assign command which assign data to a universe symbol will be parsed.

The following data will be assigned to the function name:

- Symbol pointing to a procedure. The procedure performs an extension of the universe.
- Symbol which may be used to give a message to the procedure.

The syntax of the Universe Binding Command

The syntax of the command is given in the following BNF-expressions:

$\langle$Init-universe-assign$\rangle$ ::= `assignuniverse`
$\langle$Universe-name$\rangle$, $\langle$Ext-proc$\rangle$,
$\langle$U-mess-symb$\rangle$;

The $\langle$Ext-proc$\rangle$ and $\langle$U-mess-symb$\rangle$ are defined as Scheme symbols restricted to contain the alphabetic symbols and in addition the symbols `- # : ? !` . No formal definition of the syntax will be given for these symbols.

Example a universe Binding Command

```
assignfunc UNIVERSE, extension, new-element-type;
```

2.9.3 Execute the Transitions

If the "run" command is typed, then the interpreter will begin execution of the transitions.

Each predicate within a transition will be tested (in some indeterminate order) and the first transition which has its predicate computed to "true" will be executed. All updates within the choosen transition will be executed.

Then the process described above will be repeated.

The execution will run until *no* transition can be executed, or until the user stop the process. This loop is the only loop in the system which may last forever.

Syntax of the Run command

$\langle$Run-command$\rangle$ ::= `run` $\langle$Run-integer$\rangle$; **|** `run` ;
$\langle$Run-integer$\rangle$ ::= $\langle$Integer$\rangle$

The integer given to the run commands determine how many steps (transitions) to execute before the system ask the user if she want to continue. If the number zero is given, the system will run without asking the user. If no argument is given, the user is asked if she want to continue every time a transition is to be executed.

Examples of the Run commands

```
run;
run 0;
run 6;
```

2.9.4 The Reset Command

⟨Reset-data⟩ ::= `reset`

This commands resets the data structure to the initial state.

2.9.5 Print the Statistics

If the "defstat" command is typed, then the definition statistic will be displayed.

If the "records" command is typed, all numbers used in the run time statistic is printed out along with information sufficient to determine which transition or part of transition the number refers to.

If the "runstat" command is typed, the run transitions statistic will be displayed.

Syntax of the Print the Definition Statistic Command

⟨Print-def-stat⟩	::=	`defstat`
⟨Print-run-stat-stat⟩	::=	`runstat`
⟨Print-numbered-records⟩	::=	`records`

2.9.6 Read from File

If "loadalg" is the first symbol, then expressions will be loaded from a file specified by the user.

Either a single file name or two file name may be given. In the first case the expressions will be loaded from the specified file and the output is written on the current output stream, in the second case the input data will be written to the first file specified and output data will be written to the second file specified.

The second file name will always get the extension ".msg" added to the name given by the user.

Syntax of the Load Algebra Command

The syntax of a load algebra command is given in the BNF-expression below:

⟨Loadalg⟩ ::= `loadalg` ⟨Input-output⟩ ;
⟨Input-output⟩ ::= ⟨Input-file⟩ |
⟨Input-file⟩ ⟨Output-file⟩

Example of Load Algebra Commands

```
loadalg "input-file.ea" "output-file";
loadalg "only-input-file";
```

2.9.7 Load Scheme Definitions

If "loadproc" is given, then a file which contains Scheme expressions is loaded into the the user's Lisp environment.

The intended use of this mechanism is to provide the flexibility to the user to associate Scheme procedures to function symbols and universe symbols. The Scheme procedures may compute function values, perform function updates, compute the universe value or perform the an extension of one of the universes.

Syntax of the Load Scheme Definition Command

The syntax of a load Scheme definition command is given in the BNF-expression below:

⟨Loadproc⟩ ::= `loadproc` ⟨Input-file⟩ ;

Example of a Load Scheme Definition Command

```
loadproc "procedure-file.scm";
```

2.9.8 Execute a Scheme Expression

If "scheme" is given as the first symbol, then a Scheme expression will be read and executed.

The Scheme expression will be executed in the user's Lisp environment. This mechanism makes it easy to test the user's own Scheme procedure in the user's environment.

Syntax of the Execute Scheme Expression Command

The syntax of the execute Scheme expression command is given in the BNF-expression below:

⟨Scheme-expr⟩ ::= `scheme` ⟨Scheme-expr⟩ ;

Example of an Execute Scheme Expression Command

```
scheme (+ 4 4)
```

2.9.9 Display User Bindings

If "bindings" is typed, then all bindings in the user environment will be displayed.

Syntax of the Display User Bindings Command

⟨Display-bindings⟩ ::= `bindings`

2.9.10 Fetch Data Assigned to the Function Name

If "fetchfunction" is given, then the data assigned to a function name will be printed.

The function name must be given as the second symbol.

Syntax of the Fetch Function Command

The syntax of a fetch function command is given in the BNF-expression below:

⟨Fetch-function⟩ ::= `fetchfunction` ⟨Function-name⟩ ;

Example of Fetch Function Commands

```
fetcfunction myfunction;
```

2.9.11 Fetch Data Assigned to the Universe Name

If "fetchuniverse" is given, then the data assigned to a universe name will be printed.

The universe name must be given as the second symbol.

Syntax of the Fetch Universe Command

The syntax of a fetch universe command is given in the BNF-expression below:

⟨Fetch-universe⟩ ::= `fetchuniverse` ⟨Universe⟩ ;

Example of Fetch Universe Commands

```
fetcuniverse UNIVERSE;
```

2.9.12 Load System Procedures

If "loadprog" is typed as the first symbol, then new version of procedures belonging to the evolving algebra system itself will be loaded from the given file into the Scheme environment.

This command is only intended for system maintenance, and the command can safely be discarded if no change is to be made to the system.

Syntax of the Load System Procedures Command

The syntax of a load system procedures command is given in the BNF-expression below:

⟨Loadprog⟩ ::= `loadprog` ⟨Input-file⟩ ;

Example of a Load System Procedures Command

After you have fixed an obscure bug in one of the procedures in the file "ea-updates-level.scm", you may load the the procedures into the system and try again.

```
loadprog "ea-updates-level.scm"
```

2.10 Update of statistical records

When various definitions are given to the interpreter, all the statistical records regarding the definitions will be maintained.

The following data will be updated during the process of defining the algebra and transitions:

- The number of defined algebras, signatures, functions, universes and transitions.
- The number of all defined function updates, universe updates, universe extensions and universe function updates for all transitions together and specified for each transition.
- For each universe update the number of defined universe extensions and defined universe function updates.

Chapter 3

Processing

The purpose of this chapter is to describe the internal processing of the data read into the system.

All definitions and transitions will be saved in an internal data-structure for as long the interpreter runs the session. When the user execute the transitions, the internal data-structure will be changed in order to reflect the changes in the simulated system.

3.1 The Internal Data Structure

The internal data structure is divided in to five global lists.

The global lists are divided as follows:

Definitions Four lists of definitions is made.

Transitions A list of transitions is made.

The definitions consist of lists of function definitions, universe definitions, signature definitions and algebra definitions.

The definitions of transitions constist of a list of all transitions given so far during a session.

See Figure 3.1 for an overview.

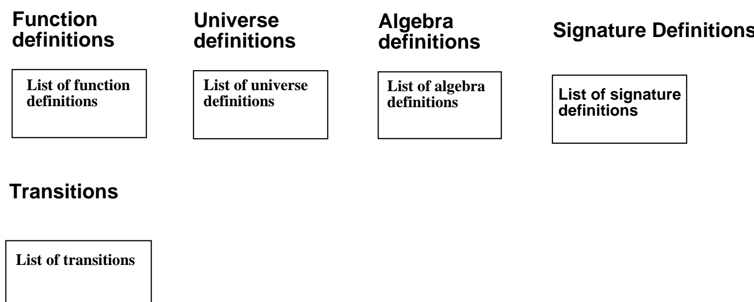

Figure 3.1: The global lists

3.1.1 List of Function Definitions

All function definitions are collected into a list.

Each function definition is in turn divided into the following items:

A function name A Scheme procedure definition is assigned to the function name. The procedure represents the function by computing the function values. More than one function definition may share the same procedure definition.

An update procedure key The update procedure key points to an Scheme procedure definition. The procedure will assign a new value to the function during the execution of updates.

A signature definition key The key points to the signature definition of the function. More than one function may share the same signature definition.

A function message key The key points to an internal message which may be used to describe the state of the the function during execution of the transitions.

A function data key The function data key points to some data. The data may be used to store the values of the function. If the function takes arguments, then the values of the arguments may be used as a key to a table of function values.

3.1.2 Universe Definitions

All universe definition are collected into a list.

Each universe definition consists of the following items:

A universe name A Scheme procedure definition is assigned to the universe name. The procedure may computer the value of all elements in the universe. More than one universe may share the same procedure definition.

An extension procedure key The extension procedure key points to an Scheme procedure definition. The procedure will extend the universe during the execution of a universe extension.

A contraction procedure key The key points to a contraction procedure definition. *The contraction update operation is not implemented. Therefore the key may point to a dummy procedure name.*

A universe message key The key points to an internal message which may be used to describe the state of the the universe during execution of the transitions.

Universe data key The universe data key points to the data, which in some way represents the values of the elements in the universe.

List of definitions

Function definition data structure (list)

Fnction name	Update procedure key	Signature definition key	Function message key	Function data key

Universe definition data structure (list)

Universe name	Extension procedure key	Contraction procedure key	Universe message key	Universe data key

Algebra definition (list)

Algebra name	Algebra list

Signature definition (list)

Signature key	Signature expression

Figure 3.2: List of definitions

3.1.3 List of Algebra Definitions

The algebra definition looks like:

An algebra name Name of an algebra.

An algebra definition The definition is a list of the universes, and function names belonging to the algebra.

3.1.4 List of Signature Definition

The signature definition consists of:

A signature key This is an unique key to the signature definition.

A signature definition An signature definition given as type to a function.

See Figure 3.2 for an overview.

3.1.5 List of Transitions

All transitions are contained in a single list.

3.1.6 A Transition

A transition has three elements:

A transition key An unique key.

A predicate A predicate expression used to test if a the updates in the transitions can be chosen for execution.

A list of updates Contains all updates belonging to a transitions.

See Figure 3.3 and 3.4 for an overview.

3.1.7 An update

An update has the following form:

An update key An unique key.

An update type Gives the type of the update object. An update may be of one of three types. The three types are function update, universe update and contraction.

An update object The form of the update object depends of the type of update.

The update object may have one of the following forms:

- A function update expression if the type of update is function update.
- A contraction expression if the type of update is contraction.
- A compound universe update object, if the type of the update is universe update. See below.

A universe update object consists of the following elements:

A list of universe extensions See below.

A list of universe function updates See below.

3.1.8 A Universe Extension

A universe extension has the following elements

A universe extension key An unique key. The key is the universe name of the extension.

The universe name Name of the universe to receive new elements.

Transitions data structure

A transition (array)

Transition key	Predicate	List of updates

All transitions are saved in a list of transitions in the user environment.

An update (array)

Update key	Update type	Update object

An update object may be of one of the following types:

1. Function update object
2. Contraction object
3. Universe update object

1. Function update object (list)

Function update expression

2. Contraction object (list)

Contraction expression

3. Universe update object (array)

List of universe extensions	List of function updates

Figure 3.3: Transition data structure

Number of new elements An expression or an integer giving the number of elements to be added to the universe. If the item is not given, the number of new elements will default to one.

A generic constant If instances of the function update for all new elements of the universe extension are to be generated, the generic constant are used as a placeholder for the instance number of each instance which is to be made. This number is part of a number expression used to compute the identifier number of the new element. The identifier number is used in the corresponding instances of the universe function update. The generic constant may be omitted if there is no need to generate instances of the universe function update for all new elements.

3.1.9 A Universe function update

A universe function update has the following elements:

A universe function update key An unique key.

A function update expression

A list of universe name The list contains all universe name to iterate over for this function update, when creating instances for all new elements. This list is empty if the specification says that there is no need to generate instances for all new elements added in the universe extension for this function update (a way to avoid duplication of instances).

A universe function updates is essential the same as the function update. The only difference is that the universe function expression may contain the expression used to compute the identifier number. The identifier number is used to identify one of the new elements added to the one of the universe or to identify all the new elements in all instances generated for this function update, if a generic constant occurs in the universe function update. Every universe used in an sub expression of the form `temp(U,Every(e))` contribute to the number of instances to be generated.

3.1.10 Statistical data

Figure 3.5 and figure 3.6 gives an overview of the way the internal data structure of the statistical data is organized.

A brief description of the data structure is given below. See section 4.2 for a detailed description of the statistical data.

The totals The sums for all transitions are collected in one record.

The list of transitions All numbers for each transition are stored in a list of statistical transition records.

The transition record Each transitions record contains statistical data regarding each transition. In addition a list of statistical universe update records will be stored, if necessary.

Elements in an universe update object

Universe extension (array)

Universe extension key	Universe name	Number of new elements	Generic constant

In list of universe extensions

Universe function update (array)

universe function update pointer	Function update expression	Array of universe names

In list of universe function updates

Figure 3.4: Elements in a universe update object

Statistical main data structure

Totals (record)

Total record

Transitions (list)

List of transition records

Figure 3.5: Top level statistical data structure

The universe update record A universe update record contains numbers regarding universe extensions and universe function updates. In addition each universe updates record contains a list of statistical universe extension records.

The universe extension record Each universe extension record contains the sum of number of elements, which adds to the universe when performing this extension.

Thr universe function update record Each record contains the number of universe functions updates performed, the the numbers of universe functions updates discarded and the number of universes to iterate over when generating instances of the universe function update.

3.2 Processing of the Initial Values

Initial values may be assigned to the functions and the universes.

List of statistical records

Transition statistical record

Transition key	List of update statistical records	Statistical record

Universe update statistical record

Universe update key	List of universe extension statistical records	List of universe update statistical records	Statistical record

Universe extension statistical record

Universe extension key	Statistical record

Universe update statistical record

Universe update key	Statistical record

Figure 3.6: Statistical data records

Initialize Function Values

Expressions which assign initial values to a function are computed in sequence (not simultaneous) before any transition is executed. Procedures defined as part of the Evolving Algebra environment are used to assign the initial values to the function. See subsection 3.4.1

Initialize universe values

Expressions which assign initial sets of values to the universes are computed in sequence (not simultaneous) before any transition is executed. Procedures defined as part of the Evolving Algebra environment are used. See subsection 3.4.1

3.3 Processing of the Transitions

The predicate of each transition is tested. Each transition which has its predicate evaluated to "true" is placed on a list of transitions which may be executed. The system will choose one of the transitions.

Since the choice should be made in a non determinate way, the actual choice of transition the system will make will not be documented. In addition the choice may be non fair [1].

If the evolving algebra to be simulated is going to be a determinate system, no more than one transition in the transition list should have its predicate evaluated to "true" at any time.

When one transition have been choiced, all updates belonging to the transition are executed simultaneous.

[1] As an undocumented feature: The first transition on the list of transition which may be executed, is chosen in the first version of the interpreter.

3.3.1 Processing of the Updates

Two types of update may be processed (processing of contraction is not implemented), the function update and the universe update.

Function Update

The expression on the right hand side giving the new value is computed. If the expression on the left hand side has any arguments, then the value of the arguments is computed. The value gives the key where the new value should be placed. Then the function update expression is stored on a temporary list to be processed in a final stage.

When the new values and argument values for all updates have been computed, all the new function values will be assigned in a final stage.

Universe Update

A universe update expression consists of

- for each universe to be extended in this universe update, a universe extensions,
- some function updates expressions.

For all universe extensions i the universe update, new elements are added for each universe to be extended. Then, all function updates within a universe update are processed. The way of processing is divided in to main cases. In the first cases each of the references to a new element in the function update refer to only one of the new elements added to the universe. So it is no need to create many equal instances of the function update so the system compute the function update one time. The other case a function update has at least one generic reference to all elements added to a universe. In this case we create instances for all possible combinations of new elements added to the universes using following algorithm:

```
for i=1 ... #U1
do
  ...
  for j=1 ... #Un
  do
      ...
      for k=1 ... #Um
      do
        Make instance(i,...,k,...,l) of the function update
      od
  ...
  od
...
od
```

The expression `#Un` is the number of elements added to the universe `Un` and `i`, `j` and `k` is the number which is substituted for the generic constant defined in the universe extension expression.

An example will clarify the processing. Suppose the following universe update is defined:

```
extend
  extenduniverse U 2 a;
  extenduniverse V 2 b;
whithupdates
  loc(temp(U,Every(a)),temp(V,Every(a))):=temp(V,Every(b))
endextend
```

The following instances of the functions updates are to be generated by the system:

```
  loc(temp(U,1),temp(V,1)):=loc(temp(V,1))
  loc(temp(U,1),temp(V,2)):=loc(temp(V,2))
  loc(temp(U,2),temp(V,1)):=loc(temp(V,1))
  loc(temp(U,2),temp(V,2)):=loc(temp(V,2))
```

The identifier number in the `temp` expression refer to one of the elements added to one of the universes.

Computation of (an instance of) a function updates within the universe update goes as follows:

1. Find all occurrences of the `temp` subexpression at right hand side and in the arguments at left hand side. Let these occurrences of the temporary expression receive as their values one of new elements added to the universe.

2. Compute arguments on the left hand side.

3. Compute the new value on the right hand side.

4. Return the computed function update expression.

The details regarding computation of values, performing function updates and universe extension are covered in the next section.

3.4 Represent the Functions

A function may be represented by an algorithm to compute the values or a table of values using the argument values as a key. The system should make no assumption with regards to how to find the function values. Since the the way to assign new value to a function (in practice) depends on how the function values are computed and stored, no assumptions can be made on how to assign new values to a function when performing updates.

Therefore the system makes two level of computations.

The system level is all computations that do not depend on:

- the computed values of functions,
- the finite sets representing the universes,
- the way new elements may be added to a universe, or
- the way the new value may be assigned to to a function when performing the function update.

All computations which depends on the interpretation of functions and universes belongs to the Evolving Algebra environment. All such computations belong to the environment level. The user of the Evolving Algebra system may define whichever Scheme procedures she want to be used in the Evolving Algebra Environment.

All such procedures will be computed in a special environment called "the Evolving Algebra environment".

The computation of a user defined procedure takes place when the value of a function or a element in the universe is computed, or when a universe extension or a function update is performed.

The task of writing the user level procedures require some skill in Scheme programming, and should not be considered as a very easy task.

3.4.1 Evolving Algebra Environment Procedures

In this section the author will describe the way the Evolving Algebra environment procedures are invoked by the system.

3.4.2 The Evolving Algebra Environment

Evolving Algebra environment Scheme procedures and data may be assigned to the function name and universe name symbols.

Three class of Evolving Algebra environment procedures may be defined:

Function lookup procedures These procedures retrieve or compute values of the function.

Function assign procedures These procedures assign (new or changed) values in the function value space.

Universe extension procedures These procedures makes and adds new elements to the universe.

Arguments Given to the Procedures

The arguments that may be given to the lookup procedures and assign procedures are in general:

function-name The name of the function using the procedure.

ea-environment The Evolving Algebra environment in Scheme (given by the evolving algebra system).

output-port The current output-port.

list-of-arguments The list of function argument values given to the procedure. The list of argument values will not be given if the function does not take arguments.

new-value New value to be assigned in one point of the function.

A Universe Extension Procedure

The arguments that may be given to the universe extension procedures are:

universe-name The name of the universe using the procedure.

output-port The current output-port.

ea-environment The Evolving Algebra environment in Scheme (given by the evolving algebra system).

type May be one of the symbols: "add-elements", "new-elements" and "initialset". See below.

optional argument May be either the list of elements to initialize the universe, or the number of elements to add to the universe.

A universe extension procedure may be called from the evolving algebra system as follows:

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type>)
```

or

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type> <optional-argument>)
```

A universe extension procedure must be prepared to handle the following cases according to the value of type argument passed to the procedure by the evolving algebra interpret:

initialset Initialize the universe with the list of elements given in the optional argument.

add-elements Add the elements to the universe data. The list of elements to be added are given in the optional argument.

new-elements Make new elements which is to be added to the universe later in the processing. If the optional argument is given, the argument is a positive integer telling how many element to create. If the optional argument is omitted, just one element is to be made. The new elements created must be returned to the system as a list of elements.

A Function Lookup Procedures

A lookup procedure will be used when the value of an expression is to be computed. The function may retrieve data stored in the user environment and is expected to return the computed function value.

A user defined procedure to lookup function values may be called as follows:

```
(<procedure-name> <list-of-arguments> <function-name>
                  <ea-environment> <output-port>)
```

or

```
(<procedure-name> <function-name> <ea-environment>
                  <output-port>)
```

if no list of argument values is given.

A Function Assign Procedures

A function assign procedure is expected to assign a new value or replace a value by a new value in the value space of the function. The new value have to be stored in the data structure associated to the function, more precisely the data structure pointed to by the function data key. See subsection 3.1.1.

A user defined procedure which assigns new values to a function may be called as follows:

```
(<procedure-name> <function-name> <new-value>
                  <ea-environment> <output-port>
                  <list-of-arguments>)
```

or

```
(<procedure-name> <function-name> <new-value>
                  <ea-environment> <output-port>)
```

if no list of argument values is given.

Universe extension Procedures

The universe extension procedure is expected to do one of the following tasks depending on the type argument given:

- Initialize the universe.
- Compute the new elements to be added to the universe.
- Add the new elements to the universe.

If the interpreter is going to initialize the universe, the call sequence will be:

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type>
                  <list-of-elements>)
```

The type is specified to be the symbol `initialset`.

If new elements are to be made, the procedure may be invoked as follows:

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type>
                  <number-of-new-elements>)
```

If only one element is to be made, the number of elements can be omitted:

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type>)
```

The type is specified to be the symbol `new-elements`.

New elements are first added to the universe in the final stage of the execution of the transition. The call sequence will be:

```
(<procedure-name> <universe-name> <output-port>
                  <ea-environment> <type>
                  <list-of-new-elements>)
```

In this case the type is specified to be the symbol `add-elements`.

Universe Lookup Procedure

No computation which lookup the set of elements in the universe are implemented.

3.5 Some Written Evolving Algebra Environment Procedures

In order to get the system to work some Evolving Algebra environment procedures have to be provided. Here follows a description of some of the Evolving Algebra environment procedures written so far. The list of the Evolving Algebra environment procedures may be extended and the procedures may be changed.

Users of the system may also write new Evolving Algebra environment procedures to handle new data structures, if needed.

The written lookup and assign procedures cover the following cases:

- The function is a constant. It is quite simple to store and retrieve the value.
- A procedure which compute the function value is initially associated to the function. A table is used to store new values which override the values computed by the procedure. A list of argument values is used as a key to the table or as arguments to the procedure.
- A table store all values in the function value space. The list of argument values is used as a key to the table.
- No function update is permitted.

The written universe extensions procedure may initialize the universe, make new elements to the universe or add the new elements to the universe in the final stage of executing the transitions.

Only the main procedures invoked directly by the system are treated below.

3.5.1 Some Lookup procedures

Below we describe the lookup procedures written.

Constant-std-lookup-data

The procedure *constant-std-lookup-data* fetch the value associated to a constant.

The procedure takes arguments as shown below:

```
(define (constant-std-lookup-data const-name env output-port)
  ...)
```

Table-std-lookup-data

The procedure *table-std-lookup* retrieve a value associated to a function of non zero arity. The values are stored in a table. The list of argument values is used as a key to the table.

The procedure takes arguments as shown below:

```
(define (table-std-lookup arg-values func-name env output-port)
   ...)
```

Comb-table-std-lookup-data

The procedure *comb-table-std-lookup* get a value associated to a function of non zero arity. Some of the function values are stored in a table, other values may be computed by another procedure associated to the functions data. The list of argument values is used as a key to the table or given as arguments to the procedure computing the value.

The procedure takes arguments as shown below:

```
(define (comb-table-std-lookup arg-values func-name env output-port)
   ...)
```

List Operations

The following additional lookup procedures which performs list operations are implemented:

- *pick-elem-from-list* Pick one element from the list.
- *concat-list* Append two lists.
- *make-list* Make a list of the arguments given.

- *first-from-list* Takes first element from the list.
- *tail-from-list* Takes the tail of the list.
- *take-nth-first-from-list* Take away the nth first elements from the list.
- *empty-list* Tests if the list is empty.
- *add-to-list* Adds a new element in head of a list.
- *length-of-the-list* Gives the length of the list.

Arithmetic Operations

The following additional lookup procedures which performs some very basic arithmetic operations are implemented:

- *add-numbers* Adds two numbers.
- *subtract-numbers* Subtracts the second number from the first.
- *multiply-numbers* Multiplies two numbers.
- *divide-numbers* Divides the first number by the second number.
- *is-number* Tests if the value is a number.

Other Operations

- *compare-two-vars* Compare two Scheme symbols.

3.5.2 The Assign Procedures

User-update-constant

The procedure *user-update-constant* assigns a new value to a constant. No list of argument values may be given to this procedure.

The procedure takes arguments as shown below:

```
(define (user-update-constant func-name value output-port env
              #!optional args-value)
   ...)
```

User-update-function

The procedure *user-update-function* will assign a new or changed value in the function value space. The procedure can handle the following cases:

- The values are stored in a table. The list of argument values is used as a key to the stored values. This cases applies if the function message symbol is set to "std-table". See Figure 3.7.

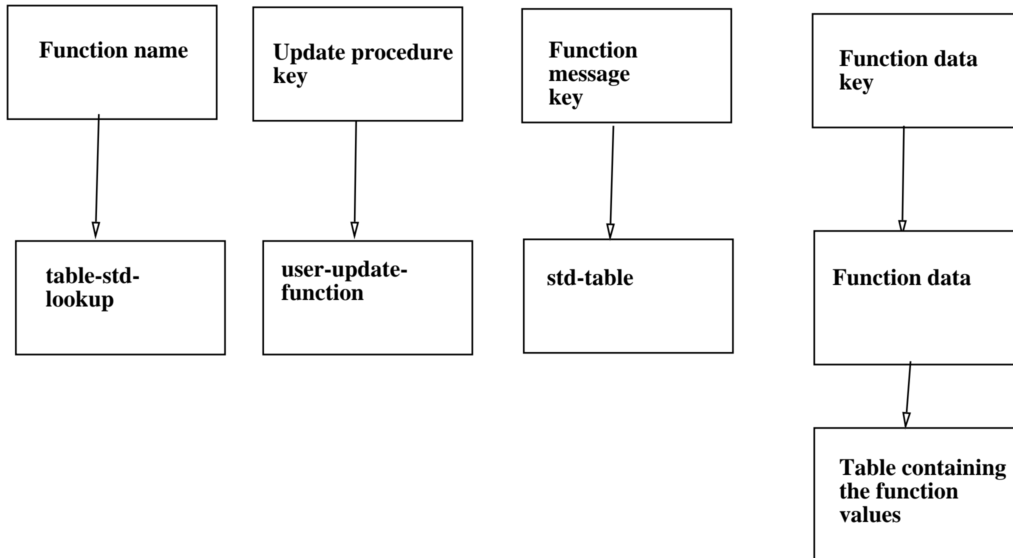

Figure 3.7: The function values are stored in a table

- Initially, a specific lookup procedure is used to compute the values of the function. The new value to be assigned will override one of the values computed by the specific lookup procedure. A table storing the new values is created. The new value is stored in the table using the list of argument values as the key. The function message symbol is set to "std-comb-table". The procedure used to compute the value is replaced by the *comb-table-std-lookup* procedure, and a pointer to the original procedure is stored in the data structure. This case applies if the function message symbol is not set to any value indicating one of the other cases. See Figure 3.8.
- A table containing assigned values is used together with the specific lookup procedure. The value to be assigned is stored in the table. The list of argument values is used as a key. This cases applies if the function message symbol is set to "std-comb-table". *The function message symbol should never be set to "std-comb-table by the user.* See Figure 3.9.
- No value is assigned, if the function message symbol is set to "upd-not-permitted". The procedure prints an error message.

The procedure takes arguments as shown below:

```
(define (user-update-function func-name value output-port env
             #!optional args-value)
```

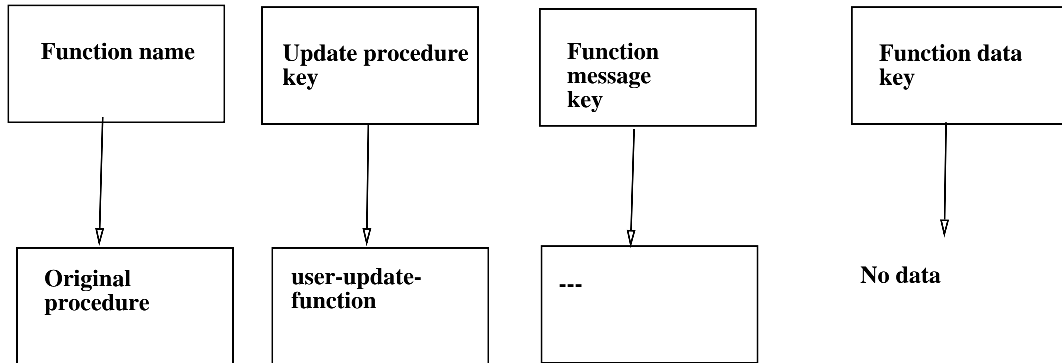

Figure 3.8: The function values are computed by a procedure

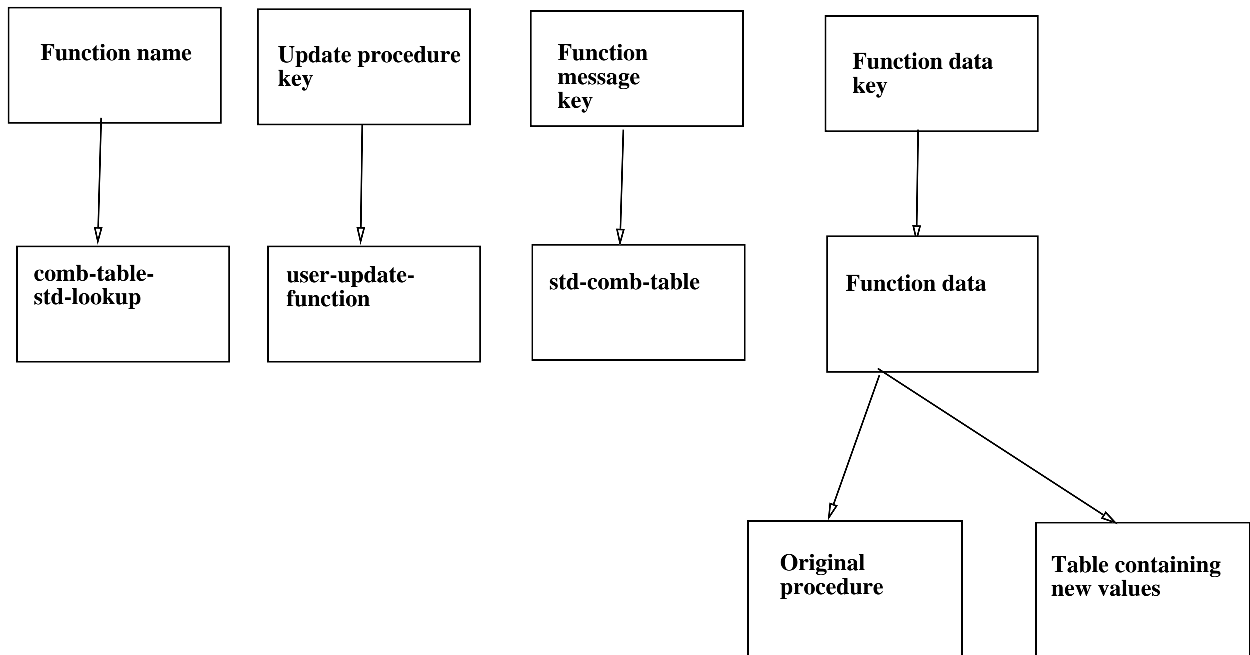

Figure 3.9: Function values computed by a procedure or stored in a table

```
   ...)
```

3.5.3 A Universe Extension Procedure

Std-ext-collection

The procedure *std-ext-collection* may perform one of the following tasks depending of the type given:

- Initialize the universe using a given list of elements, if the given type is set to "initialset".
- Make one or more new elements to be added to the universe, if the given type is "new-elements".
- Add new elements to the universe at the end of the update, if the given type is "add-elements".

If the procedure is going to make new elements, the form of new elements will vary depending on the value of the universe message data field. Three cases are implemented:

- If "number" is given as the universe message symbol, the new elements created are numbers.
- If "newsymbol" is given unique new symbols are created.
- Unique new symbols are also created if none of the cases above applies.

The procedure takes arguments as shown below:

```
(define (std-ext-collection universe output-port env type
              #!optional optional-arg)
   ...)
```

Chapter 4

Output and Statistics

4.1 Result of the Computation

4.1.1 Definitions and Commands

When executing the definitions or command the evolving algebra the interpreter will return the the definitions (in some slightly different form) or the result of the computations.

The output generated after parsing the simple stack example given in section 2.1, is given in appendix B.

4.1.2 Executing the Transitions

When executing the transitions the interpreter will track the simulation performed. In addition the output which may be generated from the user defined procedures will be printed.

The the output generated after running the simple stack example given in 2.1, is given in appendix A.

4.2 Statistics

4.2.1 Definition Statistic for an Evolving Algebra Specification

The following statistical data is collected when an evolving algebra definition is parsed.

The Function Update Defined as Part of the Universe Update

For each function update defined as part of a universe update: (also called *universe function update*):

- The number of universes participating in generating instances of this universe function update (Number of universes associated with an "Every" number expression).

pau

The difference between the maximum number of instances of universe function updates which in principle can be generated and the actual number of instances of universe function updates which needs to be generated, tells how much we profit when the system optimizes the generation of the instances. All instances which is duplicates are not generated by the interpreter.

The number of instances generated for each function updates is as shown below:

$$\prod_{i=1}^{pau} pv_i$$

where pv_i is the number of elements added to the universe i of the participating universes and pau is the number of participating universes for the universe function update. The number pau will be equal or less than the number of defined universe extensions.

If the system did not count how many universe name which was associated with an "Every" number expression in the universe function update we would have to take all defined universe extension in count, when generating instances of the function update. That would mean generating many equal instances of the universe function update.

The Universe Update

For each universe update:

- The number of defined universe extensions (which is the number of universes to be extended).

$$dn$$

- The number of defined function updates within the universe update.

$$du$$

The Transitions

For each defined transition the following sums will be displayed:

- The number of defined function updates.

$$df$$

- The number of defined universe updates.

$$dm$$

- The sum of all universe extensions defined in this transition.

$$\sum_{i=1}^{dm} dn_i$$

where dn_i is the number of universes to be extended within the universe update i and dm is the number of defined universe updates within the transition.

- The sum of all universe function updates defined in this transition.

$$\sum_{i=1}^{dm} du_i$$

where du_i is the number of universe function updates defined within universe update i and dm is the number of defined universe updates within the transition.

The Totals Printed

The following totals will be printed:

- The number of defined functions (with a signature).

$$dfn$$

- The number of defined universe names.

$$dun$$

- The number of defined transitions.

$$dk$$

- Total number of functions updates defined.

$$\sum_{g=1}^{dk} df_g$$

where df_g is the number of function updates defined in the transition g.

- Total number of universe updates defined.

$$\sum_{g=1}^{dk} dm_g$$

where m_g is the number universe updates defined in the transition g.

- Total number of defined universe extensions.

$$\sum_{g=1}^{dk} \sum_{i=1}^{dm} dn_{g,i}$$

where $dn_{g,i}$ is the number of universes to be extended within transition g and universe update i.

- Total number of defined universe function updates.

$$\sum_{g=1}^{dk} \sum_{i=1}^{dm} du_{g,i}$$

where $du_{g,i}$ is the number of universe function updates defined within transition g and universe update i.

4.2.2 Execution Statistic for an Evolving Algebra Specification

The following statistical data is collected when an evolving algebra specification is performed.

The Universe Extensions

The system will print the following data as specified below for each defined universe extension:

- Number of new elements to be added to the universe:

  $$\sum_{h=1}^{pl} pv_h$$

  where pv_h is the h'th run of the transition which has this universe extension and pl is the number of times this transition is executed.

The Universe Function Updates

For each defined universe function updates the system will print:

- The number of instances of the universe function updates made and performed:

  $$\sum_{h=1}^{pl} pu_h$$

  where pu_h is number of function updates instances performed in the h'th run of this transition. The number pl is the number of times this transitions is performed.

- The number of instances of the function updates which is made and then discarded:

  $$\sum_{h=1}^{pl} dsu_h$$

  where dsu_h is number of function updates instances discareded in the h'th run of this transition. The number pl is the number of times this transitions is performed.

The maximum number of instances which can be made of a universe function update when the transition is performed one time is:

$$\prod_{i=1}^{dn} pv_i$$

where pv_i is the number of elements added to the universe i in the universe extension in this universe update and dn is the number of defined universe extensions within this universe update.

The number of instances made is ordinary less than the maximum, because the system avoids making equal instances of the universe function updates, and because some of the generated instances may be discarded as not properly defined.

The Universe Updates

For each defined universe update the following sums will be printed:

- Total number of elements added to all the universes extended in this universe update:

$$\sum_{h=1}^{pl}\sum_{j=1}^{dn} pv_{h,j}$$

  where $pv_{h,j}$ is the number of elements to be added to the universe when performing the universe extension j in this universe update in the h'th run of this transition. The number dn is the number of defined universe extensions in the universe update and the number pl is the number of times this transition is performed.

- Total number of (instances of) universe function updates performed within this universe update:

$$\sum_{h=1}^{pl}\sum_{k=1}^{du} pu_{h,k}$$

  where $pu_{h,k}$ is the number (of instances) of universe function updates performed for the defined universe function update k in the h'th run of this transition. The number du is the number of defined universe function updates within the universe update and the number pl is the number of times this transition is performed.

- Total number of instances of universe function updates which is discarded within this universe update.

$$\sum_{h=1}^{pl}\sum_{k=1}^{du} dsu_{h,k}$$

  where $dsu_{h,k}$ is the number of instances of universe function updates discarded for the universe function update k in the h'th run of this transition. The number du is the number of defined universe function updates within the universe update and the number pl is the number of times this transition is performed.

The Transitions

For each transition defined:

- Number of times this transition is executed during the run.

$$pl$$

- Number of universe updates performed.

$$M_T = pl(dm)$$

  where pl is the number of times the transition is performed and dm is the number of defined universe updates within the transition.

- Number of function updates performed.

$$F_T = pl(df)$$

where *pl* is the number of times the transition is performed and *df* is the number of defined (simple) functions updates within the transition.

- Number of the universe extensions performed.

$$N_T = pl(\sum_{i=1}^{dm} dn_i)$$

where dn_i is the number of defined universe extensions in the universe update i, and dm is the number of defined universe updates within the transition, and *pl* is the number of time the transition is performed.

- Total number of new elements added to the all the universes. The numbers of new elements may vary each time the transition is executed, hence we use the following sum:

$$V_T = \sum_{h=1}^{pl} \sum_{i=1}^{dm} \sum_{j=1}^{dn} pv_{h,i,j}$$

where expression $v_{h,i,j}$ is the number of new elements added to the universe when performing the universe extension j in the universe update i in the execution number h of this transition. The number *pl* is the number of times this transition has been executed, the number dm is the number of defined universe updates in the transition, and dn is the number of defined universe extensions within the transition.

- Number of times a universe function update is performed.

$$PU_T = \sum_{h=1}^{pl} \sum_{i=1}^{dm} \sum_{k=1}^{du} pu_{h,i,k}$$

where $pu_{h,i,k}$ is the number of universe functions updates instances made and performed for the defined universe function update k within the universe update i in the execution number h of this transition. The number *pl* is the number of times this transition has been performed, the number dm is the number of universe updates defined within this transition and the number du is the number of universe function updates defined in this transition.

- Number of times an instance of a universe function update is discarded.

$$DU_T = \sum_{h=1}^{pl} \sum_{i=1}^{dm} \sum_{k=1}^{du} dsu_{h,i,k}$$

where $dsu_{h,i,k}$ is the number of universe functions updates instances which is made and discarded for the defined universe function update k within the universe updates i in the execution number h of the

transition. The number pl is the number of times this transition has been performed, the number dm is the number of universe updates defined within this transition and the number du is the number of universe function updates defined in this transition.

- Number of elements added to a universe divided by number of times the transition is executed.
$$V_T/pl$$
- Number of times a universe function update is performed divided by number of times the transition is executed.
$$DU_T/pl$$
- The sum of performed simple function updates and universe function updates.
$$F_T + PU_T$$

The Totals Printed

The following totals will be printed:

- Total number of times a transition is executed.
$$L = \sum_{g=1}^{dk} pl_g$$
where pl_g is the number of times the transition g is executed and dk is the number of transitions defined.
- Total number times one of the predicates is evaluated to "true".
$$P = \sum_{g=1}^{dk} pp_g$$
where dk is number of defined transitions and pp_g is number of times the predicate in the transition g is evaluated to "true".
- Total number of predicates evaluated to true divided by the total number of transitions executed.
$$P/L$$
A factor equal 1 means the total determinate system, and a number greater than 1 means some degree of indetermination in the specified evolving algebra system.
- Total number of function updates performed.
$$F = \sum_{g=1}^{dk} pl_g(df_g)$$
where pl_g is the number of times the transition g has been executed and df_g is the number of defined function updates within in the transition g.

- Total number of universe updates performed.

$$M = \sum_{g=1}^{dk} pl_g(dm_g)$$

where pl_g is the number of times the transition g has been executed and dm_g is the number of defined universe updates within in the transition g.

- Total number of universe extensions performed.

$$N = \sum_{g=1}^{dk} pl_g(\sum_{i=1}^{dm_g} dn_{g,i})$$

where pl_g is number of times the transition g has been executed and $dn_{g,i}$ is the number of defined universe extensions in the universe update i within the transition g. The number dk is the number of defined transitions, and dm_g is the number of defined universe updates within the transition g.

- Total number of new elements added to a universe.

$$V = \sum_{g=1}^{dk} \sum_{h=1}^{pl_g} \sum_{i=1}^{dm_g} \sum_{j=1}^{dn_g} pv_{g,h,i,j}$$

where $pv_{g,h,i,j}$ is the number of new elements added to the universe, when performing the universe extension j in the universe update i in the execution number h of the transition g. The number dk is the number of defined transitions, the number pl_g is the number of times the transition g is executed, dm_g is the number of defined universe updates in the transition g and dn_g is the number of defined universe extensions in the transition g.

- Total number of performed universe function updates.

$$PU = \sum_{g=1}^{dk} \sum_{h=1}^{pl_g} \sum_{i=1}^{dm_g} \sum_{k=1}^{du_g} pu_{g,h,i,k}$$

where $pu_{g,h,i,k}$ is the number of universe function updates instances made and performed for the universe function update k in the universe update i in the execution number h of the transition g. The number dk is the number of defined transitions in this specification, the number pl_g is the number of times the transition g is performed, the number dm_g is the number of defined universe updates in the transition g and the number du_g is the number of function universe updates defined within the transition g.

- Total number of discarded instances of the universe function updates.

$$DU = \sum_{g=1}^{dk} \sum_{h=1}^{pl_g} \sum_{i=1}^{dm_g} \sum_{k=1}^{du_g} dsu_{g,h,i,k}$$

where $dsu_{g,h,i,k}$ is the number of universe function updates instances made and performed for the universe function update k in the universe update i in the execution number h of the transition g. The number dk is the number of defined transitions in this specification, the number pl_g is the number of times the transition g is performed, the number dm_g is the number of defined universe updates in the transition g and the number du_g is the number of function universe updates defined within the transition g.

- The sum of function updates and universe function updates performed.

$$F + PU$$

4.2.3 Example of the statistic

The the statistics generated after running the simple stack example given in 2.1, is given in appendix C and E.

4.3 Error Handling

4.3.1 Input Errors

When the user types in a command or a definition not conforming to the syntax defined in chapter 2, then the system will return an error message and wait for a new definition or command.

If the error occurs when reading a command or definition from file, the system will simply skip that definition or command and read the next one.

4.3.2 Errors when Executing a Scheme Definitions

Executing some commands may involve invoking a Scheme evaluation in order to run a user defined Scheme procedure or executing a Scheme expression in "the user environment". Commands like "loadproc", "initial", "initialset", "scheme" and "run" will in normal cases involve execution of a Scheme expression.

If some error occurs when the systems executes a Scheme expression, the Scheme interpreter will return the error message, but will *not* halt the evolving algebra interpreter.

This way of handling Scheme errors means that the user do not need to start the system from scratch in case of such errors. Definitions stored in the memory will still remain until the user resets the internal data structure or exits from the interpreter.

On the other hand the user will not be allowed to use all Schemes debugging facilities when such error occurs. That is because this way of non-standard error handling force the system to continue in the normal Scheme interpreter mode instead of getting into the debugging mode of Scheme.

The author of the system do not consider that the benefit (eventually) gained from getting into the Scheme debugging mode when testing Scheme procedures running in "the user environment", will weigh up the annoying disadvantage of stopping the system.

Bibliography

[Bör90a] Egon Börger. A logical operational semantics of full prolog. Report 111, Wissenschaftliches Zentrum, Institut für Wissensbasierte Systeme, 1990.

[Bör90b] Egon Börger. A logical operational semantics of full prolog part II. Built-in predicates for database manipulations. In B. Rovan, editor, *MFCS'90 Mathematical Foundations of Computer Science*, number 452 in LNCS, pages 1–14. Springer, 1990.

[Bör90c] Egon Börger. A logical operational semantics of full prolog part III: Built-in predicates for files, terms, arithmetic and input-output. Report 117, Wissenschaftliches Zentrum Institut für Wissensbasierte Systeme, 1990.

[BR91a] Egon Börger and Elvinia Riccobene. A formal specification of PARLOG. Unpublished, 1991.

[BR91b] Egon Börger and Dean Rosenzweig. A formal analysis of prolog database views and their uniform implementation. To appear as: [Tech. Rep. EECS Univ. of Michigan], 1991.

[BR91c] Egon Börger and Dean Rosenzweig. From prolog algebras towards WAM — a mathematical study of implementation. To appear in: CSL'90. 4th Workshop on Computer Science Logic, Springer LNCS, 1991.

Appendix A

Execution of the Stack Example

```
Complex-predicate with value True:
("&" ("=" halt 0) ("=" (firstel cmds) "Push"))

Function update expression: (":=" cmds (next cmds))
===>
    Update function name: cmds
    New value to be assigned: ("Push" "Pop" "Stop")


Universe update
    --- Universe extensions

    Universe to be extended: STACKEL
    List of new elements to be added:
    (#[uninterned-symbol 2 new-unique-el80])

    --- Universe function updates

   Universe function update expresssion:
   (":=" stack (push stack (temp STACKEL One 1)))
   ===>
    Update function name: stack
    New value to be assigned: (#[uninterned-symbol 2 new-unique-el80]
"Temp")

Complex-predicate with value True:
("&" ("=" halt 0) ("=" (firstel cmds) "Push"))

Function update expression: (":=" cmds (next cmds))
===>
    Update function name: cmds
    New value to be assigned: ("Pop" "Stop")
```

```
Universe update
    --- Universe extensions

    Universe to be extended: STACKEL
    List of new elements to be added:
    (#[uninterned-symbol 3 new-unique-el81])

    --- Universe function updates

   Universe function update expresssion:
   (":=" stack (push stack (temp STACKEL One 1)))
   ===>
    Update function name: stack
    New value to be assigned: (#[uninterned-symbol 3 new-unique-el81]
#[uninterned-symbol 2 new-unique-el80] "Temp")

Complex-predicate with value True:
("&" ("=" halt 0) ("!" (emptystack stack)) ("=" (firstel cmds) "Pop"))

Function update expression: (":=" cmds (next cmds))
===>
    Update function name: cmds
    New value to be assigned: ("Stop")


Function update expression: (":=" stack (pop stack))
===>
    Update function name: stack
    New value to be assigned: (#[uninterned-symbol 2 new-unique-el80] "Temp")


Function update expression: (":=" value (top stack))
===>
    Update function name: value
    New value to be assigned: #[uninterned-symbol 3 new-unique-el81]


Complex-predicate with value True:
("&" ("=" halt 0) ("=" (firstel cmds) "Stop"))

Function update expression: (":=" halt 1)
===>
    Update function name: halt
    New value to be assigned: 1
```

`no-more-trans`

Appendix B

Output when Parsing the Stack Example

```
comment-line
reset-data
comment-line
comment-line
((value) VALUE)
sign
((stack) STACK)
sign
((pop) ("-->" (STACK STACK)))
sign
((top) ("-->" (STACK VALUE)))
sign
((push) ("-->" (("x" (STACK STACKEL)) STACK)))
sign
((emptystack) ("-->" (STACK BOOL)))
sign
((cmds) CMDS)
sign
((next) ("-->" (CMDS CMDS)))
sign
((firstel) ("-->" (CMDS COMMAND)))
sign
((halt) STATE)
sign
comment-line
comment-line
comment-line
loaded-user-proc
loaded-user-proc
loaded-user-proc
comment-line
loaded-user-proc
```

```
comment-line
comment-line
comment-line
(stack (constant-std-lookup-data user-update-constant std-const-dta))
(cmds (constant-std-lookup-data user-update-constant std-const-dta))
(halt (constant-std-lookup-data user-update-constant std-const-dta))
(value (constant-std-lookup-data user-update-constant std-const-dta))
(pop (tail-from-list dummy-func upd-not-perm))
(push (add-to-list dummy-func upd-not-perm))
(top (first-from-list dummy-func upd-not-perm))
(emptystack (empty-list dummy-func upd-not-perm))
comment-line
(next (tail-from-list dummy-func upd-not-perm))
(firstel (first-from-list dummy-func upd-not-perm))
comment-line
(STACK (std-ext-collection newsymbol))
(STACKEL (std-ext-collection newsymbol))
(CMDS (std-ext-collection newsymbol))
(VALUE (std-ext-collection newsymbol))
(BOOL (std-ext-collection newsymbol))
(COMMAND (std-ext-collection newsymbol))
(STATE (std-ext-collection newsymbol))
comment-line
comment-line
(":==" halt 0)
(":==" cmds #(("Push" "Push" "Pop" "Stop")))
(":==" stack #(("Temp")))
comment-line
comment-line
(":=" value (top stack))
(":=" stack (pop stack))
(":=" cmds (next cmds))
("&" ("=" halt 0) ("!" (emptystack stack)) ("=" (firstel cmds) "Pop"))
comment-line
("extend" STACKEL)
(":=" stack (push stack (temp STACKEL One 1)))
universe-update-ok
(":=" cmds (next cmds))
("&" ("=" halt 0) ("=" (firstel cmds) "Push"))
(":=" halt 1)
("&" ("=" halt 0) ("=" (firstel cmds) "Stop"))
(":=" halt 1)
("&" ("=" halt 0) (emptystack stack))
"EOF Object"
```

Appendix C

Defintition Statistic of the Stack Example

```
Statistic about Evolving Algebra definitions
--------------------------------------------

Number of defined function names (with signatures): 10
Number of defined universe names: 7
Number of defined transitions: 4
Number of defined function updates: 6
Number of defined universe updates: 1
Number of defined universe extensions: 1
Number of defined universe function updates: 1

Transition: 1
-----------
Number of defined function updates: 1
Number of defined universe updates: 0
Number of defined universe extensions: 0
Number of defined universe function updates: 0

Transition: 2
-----------
Number of defined function updates: 1
Number of defined universe updates: 0
Number of defined universe extensions: 0
Number of defined universe function updates: 0

Transition: 3
-----------
Number of defined function updates: 1
Number of defined universe updates: 1
Number of defined universe extensions: 1
Number of defined universe function updates: 1
```

```
   Universe update: 1
   ----------------
   Number of defined universe extensions: 1
   Number of defined universe function updates: 1

   For each universe function update:
   ----------------------------------

      Universe function update: 1
      -------------------------
      Number of universes which participate in creating new
      instances of the universe function update expression: 0

Transition: 4
-----------
Number of defined function updates: 3
Number of defined universe updates: 0
Number of defined universe extensions: 0
Number of defined universe function updates: 0

def-statistics
```

Appendix D

Numbers Assigned to Transitions and Updates

```
The numbers assigned to the transitions and updates
---------------------------------------------------

Note: Only function updates which is part of an universe extension
is printed.

Transition number: 1
---------------------
Predicate expression:
("&" ("=" halt 0) (emptystack stack))

Transition number: 2
---------------------
Predicate expression:
("&" ("=" halt 0) ("=" (firstel cmds) "Stop"))

Transition number: 3
---------------------
Predicate expression:
("&" ("=" halt 0) ("=" (firstel cmds) "Push"))

   Universe update number: 1
   ----------------------

      Universe extension number: 1
      -------------------------
      Universe name: STACKEL

      Universe function update-number: 1
      -------------------------------
      (":=" stack (push stack (temp STACKEL One 1)))
```

```
Transition number: 4
---------------------
Predicate expression:
("&" ("=" halt 0) ("!" (emptystack stack)) ("=" (firstel cmds) "Pop"))

all-records
```

Appendix E

Execution Statisitic of the Stack Example

```
Statistic about running Evolving Algebra
----------------------------------------

Number of times a transition is executed: 4
Number of assignments performed
    (function updates and universe function updates): 8
Number of new elements added: 2

Number of times one of the predicates is true: 4
Number of times one of the predicates is true /
    Total number of times one of the transitions is executed: 1.
Number of performed function updates: 6
Number of performed universe updates: 2
Number of performed universe extensions: 2
Number of performed universe function updates: 2
Number of of instances which could not be computed: 0

Transition: 1
-----------
Number of times this transition is performed: 0
Number of function updates performed: 0
Number of universe updates performed: 0
Number of universe extensions performed: 0
Number of new elements added to the universes: 0
Number of universe function updates performed: 0
Number of of instances which could not be computed: 0
Number of function updates and universe function updates performed: 0
Transition performed zero times. No ratios can be computed.

Transition: 2
-----------
```

```
Number of times this transition is performed: 1
Number of function updates performed: 1
Number of universe updates performed: 0
Number of universe extensions performed: 0
Number of new elements added to the universes: 0
Number of universe function updates performed: 0
Number of of instances which could not be computed: 0
Number of function updates and universe function updates performed: 1
Number of new elements / Number of times the transition is executed: 0.
Number of universe function update /
    Number of times the transition is executed: 0.

Transition: 3
-----------
Number of times this transition is performed: 2
Number of function updates performed: 2
Number of universe updates performed: 2
Number of universe extensions performed: 2
Number of new elements added to the universes: 2
Number of universe function updates performed: 2
Number of of instances which could not be computed: 0
Number of function updates and universe function updates performed: 4
Number of new elements / Number of times the transition is executed: 1.
Number of universe function update /
    Number of times the transition is executed: 1.

   Universe update: 1
   ----------------
   Number of of new elements added to the universes: 2
   Number of universe function updates performed: 2
   Number of of instances which could not be computed: 0

   Number of new elements to each universe extension:
   ------------------------------------------------

      Universe extension: 1
      -------------------
      Number of of new elements added to the universe: 2

   Number of instances generated for each universe function update:
   --------------------------------------------------------------

      Universe function update: 1
      -------------------------
      Number of of instances generated for the universe
      function update expression: 2
      Number of of instances which could not be computed: 0
```

```
Transition: 4
-----------
Number of times this transition is performed: 1
Number of function updates performed: 3
Number of universe updates performed: 0
Number of universe extensions performed: 0
Number of new elements added to the universes: 0
Number of universe function updates performed: 0
Number of of instances which could not be computed: 0
Number of function updates and universe function updates performed: 3
Number of new elements / Number of times the transition is executed: 0.
Number of universe function update /
    Number of times the transition is executed: 0.

run-time-statistics
```