Project: DRA FRONT END FILTER PROJECT

Title: Specifications of *hide* and *updateState*

Ref: DS/FMU/FEF/005 *Issue: Revision* : *4.3* *Date:* 5 June 2016

Status: Draft *Type:* Specification

Keywords:

Author:

Name	*Location*	*Signature*	*Date*
G. M. Prout	WIN01		

Authorisation for Issue:

Name	*Function*	*Signature*	*Date*
R.B. Jones	HAT Manager		

Abstract: The formal specifications of the functions *hide* and *updateState* for the DRA front end filter project RSRE 1C/6130.

Distribution: HAT FEF File
Simon Wiseman

0 DOCUMENT CONTROL

0.1 Contents List

0.2 Document Cross References

[1] DS/FMU/017. *Secure Database Technical Proposal.* High Assurance Team, ICL Secure Systems, WIN01, 21st January 1992.

[2] DS/FMU/FEF/004. *Specification of SSQL Semantics I.* G.M. Prout, ICL Secure Systems, WIN01.

[3] *Invitation to Tender RSRE 1c/6130.* DRA, December 1991.

0.3 Changes History

Issue *Revision* : *4.3* **(5 June 2016)** postfix.

Issue 4.4 Removed dependency on ICL logo font

0.4 Changes Forecast

None.

1 GENERAL

1.1 Scope

This document gives a formal specification of the functions *hide* and *updateState*. It constitutes deliverable D2 of work package 1a, as given in section 7 of the Secure Database Technical Proposal, [1].

1.2 Introduction

In the introduction to [2] we stated that the formalisation of the semantics of SSQL is to be presented in three main parts. Two of these parts, the functions *hide* and *updateState*, are concerned with the security aspects of the SSQL semantics. The functions *hide* and *updateState* are formalised in this document.

2 PRELIMINARIES

The following ProofPower instructions set up the new theory *fef005* and set the context for the proof tools.

SML

```
open_theory "fef004";
(force_delete_theory "fef005" handle _ => ());
new_theory"fef005";
push_pc "hol";
```

2.1 Representation and Abstraction Functions for *State* : *Exp*

We define the representation function, *repState*, and abstraction function, *absState*, for the type *State* : *Exp*.

HOL Constant

$$
\begin{array}{|ll}
 & \mathbf{repState} : State \to StateR; \\
 & \mathbf{absState} : StateR \to State \\
\hline
 & (\forall\ a \bullet absState\ (repState\ a) = a) \\
\wedge & (\forall r \bullet isState\ r \Rightarrow repState\ (absState\ r) = r) \\
\wedge & (\forall\ a_1\ a_2 \bullet repState\ a_1 = repState\ a_2 \Leftrightarrow a_1 = a_2) \\
\wedge & (\forall\ r_1\ r_2 \bullet\ (isState\ r_1 \wedge isState\ r_2) \Rightarrow \\
 & \qquad (absState\ r_1 = absState\ r_2 \Leftrightarrow r_1 = r_2)) \\
\wedge & (\forall s \bullet isState\ (repState\ s))
\end{array}
$$

3 THE FUNCTION *hide*

The purpose of the *hide* function is to return a version of the database that does not include any information the existence of which the user is not cleared to know, nor does it contain the true value of any field which the user is not cleared to see. This is achieved as follows :

Section 3.1 A function *cleanTable* is defined which takes a table and a clearance and returns a table which has been cleaned in the following way: A table that the user is not permitted to access is replaced by an 'empty' table; otherwise, all the rows and columns about whose existence the user is not cleared to know are removed, and all the data the user is not cleared to see is replaced by dummy data.

Section 3.2 A function *cleanDirectory* is defined which takes a directory and a clearance and returns a directory which has been cleaned in the following way: if the user is not permitted to know of the existence of the tables in the directory then a directory with no tables in it is returned; otherwise the tables in the directory are cleaned by *cleanTable*.

Section 3.3 Finally, the function *hide* is defined which takes a state and a clearance and returns a state which has been cleaned in the following way: all directories about whose existence the user is not cleared to know are removed and the rest are cleaned by *cleanDirectory*.

3.1 Cleaning a Table

In the case where the user is permitted to know of the existence of a table, but is not permitted to access it, i.e. his clearance does not dominate the table class, an 'empty' table is left in the directory in place of that table.

All other tables are cleaned in the following way

- Remove from the column constraints those groups of columns about whose existence the user is not cleared to know.
- Remove any columns about whose existence the user is not cleared to know.
- Remove any rows about whose existence the user is not cleared to know.
- Remove data from columns of remaining rows about whose existence the user is not cleared to know.
- Replace data that the user is not cleared to see by dummy values.

3.1.1 Clean Column Constraints

A function that takes a clearance and a table and returns a pair: the column constraints for columns whose existence is dominated by the clearance, and the corresponding set of columns.

HOL Constant

cleanColCons : $Class \rightarrow TableSpec \rightarrow ((Num \leftrightarrow ColCon) \times ColSpec\ \mathbb{P})$

$\forall\ clear : Class;\ ts : TableSpec$

$\bullet$ $cleanColCons\ clear\ ts$

$=$

$let \quad ccs = TS\_cons\ ts \rhd \{cc | clear\ dominates\ (CC\_exist\ cc)\}$

in

$(ccs, \{col : ColSpec | (CS\_consGroup\ col) \in Dom\ ccs\})$

3.1.2 Filter all Data in a Row

Filter out the data in columns about whose existence the user is cleared to know.

HOL Constant

filterRow : $ColSpec\ \mathbb{P} \rightarrow (Num \leftrightarrow Data) \rightarrow (Num \leftrightarrow Data)$

$\forall\ cols : ColSpec\ \mathbb{P};\ ds : Num \leftrightarrow Data$

$\bullet$ $filterRow\ cols\ ds$

$=$

$\{n | \exists c \bullet c \in cols \land CS\_posn\ c = n\} \lhd ds$

3.1.3 Replace Data in a Row with Dummy Data

Data consists of a classification and an item of data. An item of data that a user is not cleared to see is replaced by a dummy string *Hidden* (its classification remains).

HOL Constant

Hidden : *String*

T

HOL Constant

dummyVal : *Val*

$dummyVal = StringVal\ Hidden$

A function to replace the value of a piece of data that the user is not cleared to see by a dummy value.

HOL Constant

replaceData : $Class \rightarrow Data \rightarrow Data$

$\forall\ clear : Class;\ d : Data$
$\bullet\quad replaceData\ clear\ d$
$=$
$if\ \neg(clear\ dominates\ (Dat\_class\ d))$
$then\quad MkData\ (Dat\_class\ d)$
$\qquad (ValuedItemItem\ (MkValuedItem\ worthless\ dummyVal))$
$else\quad d$

3.1.4 Clean all Data in a Row

Filter out the data in columns about whose existence the user is cleared to know. Replace data in fields that the user is not cleared to see by dummy values.

HOL Constant

cleanRow : $Class \rightarrow ColSpec\ \mathbb{P} \rightarrow Row \rightarrow Row$

$\forall\ clear : Class;\ cols : ColSpec\ \mathbb{P};\ r: Row$
$\bullet\quad cleanRow\ clear\ cols\ r$
$=$
$MkRow\ (R\_exist\ r)((filterRow\ cols\ (R\_data\ r))\ ⨾\ Graph(replaceData\ clear))$

3.1.5 Clean all Rows in a Table

Delete all rows about whose existence the user is not cleared to know, and then clean remaining rows.

HOL Constant

cleanRows : $Class \rightarrow ColSpec\ \mathbb{P} \rightarrow Row\ LIST \rightarrow Row\ LIST$

$\forall\ clear : Class;\ cols : ColSpec\ \mathbb{P};\ rs: Row\ LIST$
$\bullet\quad cleanRows\ clear\ cols\ rs$
$=$
$Map\ (cleanRow\ clear\ cols)\ (rs \upharpoonright \{r{:}Row | clear\ dominates\ (R\_exist\ r)\})$

3.1.6 The function *cleanTable*

The function *cleanTable* either returns an 'empty' table or a table where the rows and columns that the user is not cleared to see have been removed and fields that the user is not cleared to see have been overwritten by dummy fields.

HOL Constant

cleanTable : $Class \rightarrow TableSpec \rightarrow TableSpec$

$\forall\ clear : Class;\ ts{:}TableSpec$
$\bullet$ $cleanTable\ clear\ ts$
$=$
if $\neg(clear\ dominates\ (TS\_class\ ts))$
then $MkTableSpec$
$(TS\_class\ ts)$
$(TS\_class\ ts)$
$\{\}\{\}[]$
else $let\ (ccs,cols) = cleanColCons\ clear\ ts$
in
$MkTableSpec$
$(TS\_class\ ts)$
$(TS\_maxRow\ ts)$
$cols$
ccs
$(cleanRows\ clear\ cols\ (TS\_rows\ ts))$

3.2 Cleaning a Directory

Any tables in a directory about whose existence the user is not cleared to know can be removed from that directory. These are all the tables in a directory whose *class* component is not dominated by the user's clearance (i.e. the user is cleared to know the existence of the directory, but is not cleared to see the structure of the directory). To clean a directory, the tables in that directory will either be hidden or modified by *cleanTable*.

HOL Constant

cleanDirectory : $Class \rightarrow Directory \rightarrow Directory$

$\forall\ clear : Class;\ dir : Directory$
$\bullet$ $cleanDirectory\ clear\ dir$
$=$
$MkDirectory$
$(if \quad (clear\ dominates\ (Dir\_class\ dir))$
$then \quad (Dir\_tables\ dir \fatsemi Graph(cleanTable\ clear))$
$else \quad \{\})$
$(Dir\_exist\ dir)$
$(Dir\_class\ dir)$

3.3 Cleaning a State

Finally, any directories about whose existence the user is not cleared to know can be removed from the state. All remaining directories will be cleaned by *cleanDirectory*. First, we define *hideR* on the representation state.

HOL Constant

hideR : $Class \times StateR \rightarrow StateR$

$\forall\ clear : Class;\ s : StateR$
$\bullet$ $hideR(clear,s)$
$=$
$(s \rhd \{dir \mid clear\ dominates\ (Dir\_exist\ dir)\}) \fatsemi Graph(cleanDirectory\ clear)$

Now the function *hide* on *State* : *Exp*.

HOL Constant

hide : $Class \times State \rightarrow State$

$\forall\ clear : Class;\ s : State$
$\bullet$ $hide(clear,s) = absState(hideR(clear,\ repState\ s))$

4 THE FUNCTION *updateState*

In [2] the function *processQuery* is defined. This function takes a query, a user's classification and a state and returns the result of the query, i.e., the output to be seen by the user, an update to be applied to the data base and some error messages. The effect of a query depends on the type of the query, which may be an *Insert*, a *Delete* an *UpDate* or a *Select*.

We have not yet determined whether it is necessary to have as a state invariant that column existence classifications are in ascending order.

4.1 Auxiliary Functions

When a user makes an update request on a row, the position of the row in a table will be its position after rows about whose existence the user is not cleared to know have been hidden. We require a function that reveals the true row position given the hidden row position.

HOL Constant

revealRow : $Class \rightarrow TableSpec \rightarrow Num \leftrightarrow Num$

$\forall\ clear: Class;\ ts : TableSpec$

$\bullet$ $revealRow\ clear\ ts$

$=$

$let \quad visiblerows = \{r{:}Row | clear\ dominates\ (R\_exist\ r)\}$

in

$Squash(Id(Dom((ListRel(TS\_rows\ ts)) \rhd visiblerows)))$

We provide the auxiliary functions *replaceRows* and *changeSpec* from Annex 2 *SSQL Abstract Machine* of the ITT [3].

HOL Constant

replaceRows : $TableSpec \rightarrow Row\ LIST \rightarrow TableSpec$

$\forall\ ts : TableSpec;\ rs : Row\ LIST$

$\bullet$ $replaceRows\ ts\ rs$

$=$

$MkTableSpec(TS\_class\ ts)(TS\_maxRow\ ts)(TS\_colspecs\ ts)(TS\_cons\ ts)\ rs$

The operator @ is the partial function application operator.

HOL Constant

changeSpec : $Tab \rightarrow TableSpec \rightarrow StateR \rightarrow StateR$

$\forall\ i : Tab;\ ts : TableSpec;\ s : StateR$

$\bullet$ $changeSpec\ i\ ts\ s$

$=$

$let \quad dir = s\ @\ Front\ i$

in

$let \quad dir' = MkDirectory(Dir\_tables\ dir \oplus \{(Last\ i, ts)\})$
$(Dir\_exist\ dir)(Dir\_class\ dir)$

in

$s \oplus \{(Front\ i, dir')\}$

A function that returns the visible columns of a table.

HOL Constant

visibleCols : $Class \rightarrow TableSpec \rightarrow ColSpec\ \mathbb{P}$

$\forall\ clear : Class;\ ts : TableSpec$

$\bullet$ $visibleCols\ clear\ ts$

$=$

$Snd(cleanColCons\ clear\ ts)$

A function *tabExists* is given that determines whether or not a table name exists at a particular clearance in a particular state.

HOL Constant

tabExists : $Class \rightarrow Tab \rightarrow StateR \rightarrow Bool$

$\forall\ c : Class;\ i : Tab;\ s : StateR$

$\bullet$ $tabExists\ c\ i\ s$

$\Leftrightarrow$

$Front\ i \in Dom\ s$

$\wedge$

$c\ dominates\ Dir\_exist\ (s\ @\ Front\ i)$

$\wedge$

$c\ dominates\ Dir\_class\ (s\ @\ Front\ i)$

$\wedge$

$Last\ i \in Dom\ (Dir\_tables\ (s\ @\ Front\ i))$

The function *getTable* returns a table given its full table name. This will only be applied in the case where the table name exists.

HOL Constant

getTable : $Tab \rightarrow StateR \rightarrow TableSpec$

$\forall\ i : Tab;\ s : StateR$

$\bullet$

$getTable\ i\ s = (Dir\_tables(s\ @\ (Front\ i)))\ @\ (Last\ i)$

4.2 Insert

The function *colDefaults* takes a partial function from column number to data and a table and adds the appropriate column number - default value pairs which correspond to the columns that were hidden to the user. The default values are obtained from the table information.

HOL Constant

colDefaults : $Class \to TableSpec \to (Num \leftrightarrow Data) \to (Num \leftrightarrow Data)$

$\forall\ clear : Class;\ ts : TableSpec;\ ds : Num \leftrightarrow Data$
$\bullet$ $colDefaults\ clear\ ts\ ds$
$=$
$ds \oplus \{(n,d) \mid \exists c \bullet \neg\ c \in visibleCols\ clear\ ts \wedge n = CS\_posn\ c \wedge d = CS\_default\ c\}$

An *ambiguousColumn* error is returned if the data in a row to be inserted is not a partial function.

HOL Constant

insertQuery : $(Class \times Insert \times StateR \times TableSpec) \to StateR \times Errors$

$\forall\ clear{:}Class;\ i{:}Tab;\ ds : (Num \leftrightarrow Data)\ LIST;\ s{:}StateR;\ ts : TableSpec$
$\bullet$ $insertQuery\ (clear,(i,ds),s,ts)$
$=$
$let\ rl = Map\ ((MkRow\ clear)\ o\ (colDefaults\ clear\ ts))\ ds$
in
$if\ \neg\ Elems\ rl \subseteq RowS$
$then\ (s,[ambiguousColumn])$
$else$
$((changeSpec\ i\ (replaceRows\ ts\ ((TS\_rows\ ts) \frown rl))\ s),[])$

4.3 Delete

Delete all rows specified by *DeleteEffect*. The row numbers supplied by *DeleteEffect* correspond to the users view of the hidden state of the data base. In order to delete the correct rows from the true state of the data base, it is necessary to take into account those rows about whose existence the user is not cleared to know. If the user were not permitted access to the table then the structure of the table would have been hidden from him and therefore the request would have failed.

HOL Constant

deleteQuery : $(Class \times Delete \times StateR \times TableSpec) \rightarrow StateR$

$\forall\ clear{:}Class;\ i{:}Tab;\ ns : Num\ \mathbb{P};\ e{:}Errors;\ s{:}StateR;\ ts{:}TableSpec$

$\bullet$ $deleteQuery(clear,(i,ns),s,ts)$

$=$

$let \quad ns' = (revealRow\ clear\ ts)\ Image\ ns$

in

$let\ ns'' = ns' \cap \{i | R\_exist(Nth(TS\_rows\ ts)i) = clear\}$

$in \quad let\ rs = Extract\ (1\ ..\ Length\ (TS\_rows\ ts) \setminus ns'')(TS\_rows\ ts)$

in

$(changeSpec\ i\ (replaceRows\ ts\ rs)\ s)$

4.4 Update

Perform the updates specified by *UpdateEffect*, provided that they do not violate the security policy. A user may change the value in a field (classification stays the same) provided the field's classification dominates the user's clearance. A user may downgrade the classification of a field providing that the value in the field is overwritten.

Firstly, consider an update request on a single field of data. This will either yield a new piece of data or an error message.

HOL Constant

updateField : $Class \rightarrow Class \rightarrow (Update \times Data) \rightarrow (Data + Error)$

$\forall\ clear\ table\_class{:}Class;\ table\_d{:}Data;\ u{:}Update$

$\bullet$ $updateField\ clear\ table\_class\ (u,table\_d)$

$=$

$if \quad clear = table\_class$

$then \quad if \quad isItem\ u$

$then \quad giveVal(MkData\ (Dat\_class\ table\_d)(destItem\ u))$

$else\ if\ isClass\ u$

$then \quad if\ (destClass\ u)\ dominates\ (Dat\_class\ table\_d)$

$then \quad giveVal(MkData\ (destClass\ u)\ (Dat\_item\ table\_d))$

$else \quad giveError\ downGrade$

$else \quad giveVal(destData\ u) \quad (*\ must\ be\ Data\ *)$

$else\ if\ isItem\ u$

$then \quad if \quad (Dat\_class\ table\_d)\ dominates\ clear$

$then \quad giveVal(MkData\ (Dat\_class\ table\_d)\ (destItem\ u))$

$else \quad giveError\ underClassified$

$else \quad giveError\ classChange$

Now the effect of an update on a row.

Updates for a row must be functional otherwise an *ambiguousUpdate* error is returned. A new row is only returned if all updates are secure; otherwise a sequence of errors is returned.

HOL Constant

$$\textbf{updateRow} : \mathit{Class} \to \mathit{Class} \to ((\mathit{Num} \leftrightarrow \mathit{Update}) \times \mathit{Row}) \to (\mathit{Row} + \mathit{Errors})$$

$$\forall\ \mathit{clear}\ \mathit{table\_class}{:}\mathit{Class};\ \mathit{us}{:}\mathit{Num} \leftrightarrow \mathit{Update};\ r{:}\mathit{Row}$$

$$\bullet\quad \mathit{updateRow}\ \mathit{clear}\ \mathit{table\_class}\ (\mathit{us},r) = \mathit{if}\ \neg\ \mathit{us} \in \mathit{Functional}\ \mathit{then}\ \mathit{giveError}\ [\mathit{ambiguousUpdate}]\ \mathit{else}\ \mathit{let}\ \mathit{us}' = \mathit{RelCombine}\ \mathit{us}\ (\mathit{R\_data}\ r)\ ⨾\ \mathit{Graph}(\mathit{updateField}\ \mathit{clear}\ \mathit{table\_class})\ \mathit{in}\ \mathit{let}\ \mathit{es} = (\mathit{us}' \rhd \{x | \mathit{isError}\ x\})\ ⨾\ \mathit{Graph}\ \mathit{destError}\ \mathit{in}\ \mathit{if}\ \mathit{es} = \{\}\ \mathit{then}\ \mathit{giveVal}(\mathit{MkRow}\ (\mathit{R\_exist}\ r)((\mathit{R\_data}\ r) \oplus (\mathit{us}' ⨾ \mathit{Graph}\ \mathit{destVal})))\ \mathit{else}\ \mathit{giveError}\ (\mathit{RelList}(\mathit{Squash}\ \mathit{es}))$$

Finally, the *updateQuery* function. Updates must be functional otherwise an *ambiguousUpdate* error is returned. A check is made to prevent access to hidden columns.

HOL Constant

$$\textbf{updateQuery} : (Class \times UpDate \times StateR \times TableSpec) \rightarrow StateR \times Errors$$

$$\begin{array}{l}
\forall\ clear{:}Class;\ i{:}Tab;\ us : Num \leftrightarrow (Num \leftrightarrow Update);\ s{:}StateR;\ ts : TableSpec \\
\bullet \quad updateQuery\ (clear,(i,us),s,ts) \\
\quad = \\
\quad let\ colnums = \{n \mid \exists\ c \bullet c \in visibleCols\ clear\ ts \wedge CS_posn\ c = n\} \\
\quad in \\
\quad if \quad \neg\ us \in Functional \\
\quad then \quad (s,[ambiguousUpdate]) \\
\quad else\ if\ \neg((Dom\ (\bigcup\ (Ran\ us))) \subseteq colnums) \\
\quad then \quad (s,[noSuchColumn]) \\
\quad else \quad let \quad us' = (revealRow\ \ clear\ ts)^{\sim}\ \fatsemi\ us \\
\qquad\qquad in \\
\qquad\qquad let \quad pr = RelCombine\ us'\ (ListRel(TS_rows\ ts)) \\
\qquad\qquad\qquad\qquad \fatsemi\ Graph(updateRow\ clear\ (TS_class\ ts)) \\
\qquad\qquad in \\
\qquad\qquad let \quad es = (pr \rhd \{x | isError\ x\})\ \fatsemi\ Graph\ destError \\
\qquad\qquad in \quad if \quad es = \{\} \\
\qquad\qquad\qquad\qquad then \quad let \quad rs = RelList(ListRel(TS_rows\ ts) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \oplus\ (pr\ \fatsemi\ Graph\ destVal)) \\
\qquad\qquad\qquad\qquad\qquad\qquad in \quad (changeSpec\ i(replaceRows\ ts\ rs)\ s,[]) \\
\qquad\qquad\qquad\qquad else \quad (s,Flat(RelList(Squash\ es)))
\end{array}$$

4.5 Specification of *updateState*

The function *updateState* operates on the output from the function *processQuery*, defined in [2]. This output is a pair consisting of the effect of the update query, which depends on whether the query was a select, insert, delete or update, and a sequence of errors messages. For a select query, data is returned to the user, together with any error messages. The state of the database remains unchanged. For an insert, delete or update query, if a table name is returned by *processQuery* that does not exist, then the state of the database remains unchanged and an appropriate error message is returned. If errors were generated by *processQuery* then the state of the database remains unchanged and the errors are returned. If a valid table name and no errors are returned by *processQuery*, then if the clearance of the user does not dominate the classification of the table, again the state of the database remains unchanged and an appropriate error message is returned. Otherwise, the appropriate update to the state of the database, given by *insertQuery*, *deleteQuery* and *updateQuery*, is carried out after the security checks have been made. In the case of an insert or update query, if the security checks identified that the output of *processQuery* would result in an insecure update to the state of the database, then the state of the database remains unchanged and error messages generated by the security checks are returned to the user.

First, we define *updateStateR* on the representation state.

HOL Constant

updateStateR : $Class \times (Effect \times Errors) \times StateR \rightarrow StateR \times (Class \times (Data\ LIST\ LIST \times Errors))$

$\forall\ clear{:}Class; ef : Effect;\ es : Errors;\ s : StateR$

$\bullet$ $updateStateR(clear,(ef,es),s)$

$=$

$if\ \ isSelect\ ef$

$then\ \ (s,(clear,(destSelect\ ef,es)))$

$else\ if\ \neg(es = [])\ \ (*\ delete,update\ and\ insert\ only\ valid\ if\ no\ errors\ *)$

$then\ \ (s,(clear,([],es)))$

$else\ \ let\ \ i = tabFromEffect\ ef$

in

$if\ \ tabExists\ clear\ i\ s$

$then\ \ let\ ts = getTable\ i\ s$

in

$if\ \ \neg\ (clear\ dominates\ TS\_class\ ts)$

$then\ \ (s,(clear,([],[accessDenied])))$

$else$

$let\ \ (s',es')$

$=$

$if\ \ isInsert\ ef$

$then\ \ insertQuery(clear,destInsert\ ef,s,ts)$

$else\ if\ isDelete\ ef$

$then\ \ (deleteQuery(clear,destDelete\ ef,s,ts),[])$

$else\ \ updateQuery(clear,destUpdate\ ef,s,ts)$

in

$(s',(clear,([],es')))$

$else\ \ (s,(clear,([],[noSuchTable])))\ \ (*\ Invalid\ table\ name\ *)$

Now the function *updateState* on *State* : *Exp*.

HOL Constant

updateState : $Class \times (Effect \times Errors) \times State \rightarrow State \times (Class \times (Data\ LIST\ LIST \times Errors))$

$\forall\ clear{:}Class; ef : Effect;\ es : Errors;\ s : State$

$\bullet$ $updateState(clear,(ef,es),s)$

$=$

$let\ \ (s_r,out) = updateStateR(clear,(ef,es),repState\ s)$

$in\ \ (absState\ s_r,out)$

5 CLOSING DOWN

The following ProofPower instruction restores the previous proof context.

SML
```
pop_pc();
```

6 THE THEORY fef005

6.1 Parents

fef004

6.2 Children

fef014

6.3 Constants

absState $StateR \to State$
repState $State \to StateR$
cleanColCons $Class \to TableSpec \to Worth \leftrightarrow ColCon \times ColSpec\ \mathbb{P}$
filterRow $ColSpec\ \mathbb{P} \to Worth \leftrightarrow Data \to Worth \leftrightarrow Data$
Hidden Ide
dummyVal Val
replaceData $Class \to Data \to Data$
cleanRow $Class \to ColSpec\ \mathbb{P} \to Row \to Row$
cleanRows $Class \to ColSpec\ \mathbb{P} \to Row\ LIST \to Row\ LIST$
cleanTable $Class \to TableSpec \to TableSpec$
cleanDirectory
$Class \to Directory \to Directory$
hideR $Class \times StateR \to StateR$
hide $Class \times State \to State$
revealRow $Class \to TableSpec \to Worth \leftrightarrow Worth$
replaceRows $TableSpec \to Row\ LIST \to TableSpec$
changeSpec $Tab \to TableSpec \to StateR \to StateR$
visibleCols $Class \to TableSpec \to ColSpec\ \mathbb{P}$
tabExists $Class \to Tab \to StateR \to Bool$
getTable $Tab \to StateR \to TableSpec$
colDefaults $Class \to TableSpec \to Worth \leftrightarrow Data \to Worth \leftrightarrow Data$
insertQuery
$Class \times Insert \times StateR \times TableSpec \to StateR \times Errors$
deleteQuery
$Class \times Delete \times StateR \times TableSpec \to StateR$
updateField $Class \to Class \to Update \times Data \to Data + Worth$
updateRow $Class \to Class \to Worth \leftrightarrow Update \times Row \to Row + Errors$
updateQuery
$Class \times UpDate \times StateR \times TableSpec \to StateR \times Errors$
updateStateR $Class \times (Effect \times Errors) \times StateR$
$\to StateR \times Class \times Select \times Errors$
updateState $Class \times (Effect \times Errors) \times State$
$\to State \times Class \times Select \times Errors$

6.4 Definitions

repState
absState $\vdash$ *ConstSpec*
(λ (*repState′*, *absState′*)
• (∀ *a*• *absState′* (*repState′ a*) = *a*)
∧ (∀ *r*
• *isState r* ⇒ *repState′* (*absState′ r*) = *r*)
∧ (∀ a_1 a_2
• *repState′* a_1 = *repState′* a_2 ⇔ a_1 = a_2)
∧ (∀ r_1 r_2
• *isState* r_1 ∧ *isState* r_2
⇒ (*absState′* r_1 = *absState′* r_2
⇔ r_1 = r_2))
∧ (∀ *s*• *isState* (*repState′ s*)))
(*repState*, *absState*)

cleanColCons $\vdash$ ∀ *clear ts*
• *cleanColCons clear ts*
= (*let ccs*
= *TS_cons ts*
▷ {*cc*|*clear dominates CC_exist cc*}
in (*ccs*, {*col*|*CS_consGroup col* ∈ *Dom ccs*}))

filterRow $\vdash$ ∀ *cols ds*
• *filterRow cols ds*
= {*n*|∃ *c*• *c* ∈ *cols* ∧ *CS_posn c* = *n*} ◁ *ds*

Hidden $\vdash$ *T*

dummyVal $\vdash$ *dummyVal* = *StringVal Hidden*

replaceData $\vdash$ ∀ *clear d*
• *replaceData clear d*
= (*if* ¬ *clear dominates Dat_class d*
then
MkData
(*Dat_class d*)
(*ValuedItemItem*
(*MkValuedItem worthless dummyVal*))
else d)

cleanRow $\vdash$ ∀ *clear cols r*
• *cleanRow clear cols r*
= *MkRow*
(*R_exist r*)
(*filterRow cols* (*R_data r*)
⨾ *Graph* (*replaceData clear*))

cleanRows $\vdash$ ∀ *clear cols rs*
• *cleanRows clear cols rs*
= *Map*
(*cleanRow clear cols*)
(*rs* ⨾ {*r*|*clear dominates R_exist r*})

cleanTable $\vdash$ ∀ *clear ts*

```
    • cleanTable clear ts
        = (if ¬ clear dominates TS_class ts
           then
             MkTableSpec
               (TS_class ts)
               (TS_class ts)
               {}
               {}
               []
           else
             (let (ccs, cols) = cleanColCons clear ts
                in MkTableSpec
                     (TS_class ts)
                     (TS_maxRow ts)
                     cols
                     ccs
                     (cleanRows clear cols (TS_rows ts))))
```

cleanDirectory

```
⊢ ∀ clear dir
  • cleanDirectory clear dir
      = MkDirectory
        (if clear dominates Dir_class dir
           then
             Dir_tables dir ⨾ Graph (cleanTable clear)
           else {})
        (Dir_exist dir)
        (Dir_class dir)
```

hideR

```
⊢ ∀ clear s
  • hideR (clear, s)
      = (s ▷ {dir|clear dominates Dir_exist dir})
        ⨾ Graph (cleanDirectory clear)
```

hide

```
⊢ ∀ clear s
  • hide (clear, s)
      = absState (hideR (clear, repState s))
```

revealRow

```
⊢ ∀ clear ts
  • revealRow clear ts
      = (let visiblerows
             = {r|clear dominates R_exist r}
         in Squash
           (Id
              (Dom
                 (ListRel (TS_rows ts)
                    ▷ visiblerows))))
```

replaceRows

```
⊢ ∀ ts rs
  • replaceRows ts rs
      = MkTableSpec
        (TS_class ts)
        (TS_maxRow ts)
```

$(TS\_colspecs\ ts)$
$(TS\_cons\ ts)$
rs

changeSpec $\vdash \forall\ i\ ts\ s$
• $changeSpec\ i\ ts\ s$
$= (let\ dir = s\ @\ Front\ i$
$in\ let\ dir'$
$= MkDirectory$
$(Dir\_tables\ dir \oplus \{(Last\ i,\ ts)\})$
$(Dir\_exist\ dir)$
$(Dir\_class\ dir)$
$in\ s \oplus \{(Front\ i,\ dir')\})$

visibleCols $\vdash \forall\ clear\ ts$
• $visibleCols\ clear\ ts = Snd\ (cleanColCons\ clear\ ts)$

tabExists $\vdash \forall\ c\ i\ s$
• $tabExists\ c\ i\ s$
$\Leftrightarrow Front\ i \in Dom\ s$
$\wedge\ c\ dominates\ Dir\_exist\ (s\ @\ Front\ i)$
$\wedge\ c\ dominates\ Dir\_class\ (s\ @\ Front\ i)$
$\wedge\ Last\ i \in Dom\ (Dir\_tables\ (s\ @\ Front\ i))$

getTable $\vdash \forall\ i\ s$
• $getTable\ i\ s = Dir\_tables\ (s\ @\ Front\ i)\ @\ Last\ i$

colDefaults $\vdash \forall\ clear\ ts\ ds$
• $colDefaults\ clear\ ts\ ds$
$= ds$
$\oplus \{(n,\ d)$
$|\exists\ c$
• $\neg\ c \in visibleCols\ clear\ ts$
$\wedge\ n = CS\_posn\ c$
$\wedge\ d = CS\_default\ c\}$

insertQuery
$\vdash \forall\ clear\ i\ ds\ s\ ts$
• $insertQuery\ (clear,\ (i,\ ds),\ s,\ ts)$
$= (let\ rl$
$= Map$
$(MkRow\ clear\ o\ colDefaults\ clear\ ts)$
ds
$in\ if\ \neg\ Elems\ rl \subseteq RowS$
$then\ (s,\ [ambiguousColumn])$
$else$
$(changeSpec$
i
$(replaceRows\ ts\ (TS\_rows\ ts\ @\ rl))$
$s,\ [])$

deleteQuery
$\vdash \forall\ clear\ i\ ns\ e\ s\ ts$
• $deleteQuery\ (clear,\ (i,\ ns),\ s,\ ts)$
$= (let\ ns' = revealRow\ clear\ ts\ Image\ ns$

```
                in let ns''
                       = ns'
                         ∩ {i
                         |R_exist (Nth (TS_rows ts) i) = clear}
                   in let rs
                          = Extract
                            (1 .. # (TS_rows ts) \ ns'')
                            (TS_rows ts)
                      in changeSpec i (replaceRows ts rs) s)
updateField   ⊢ ∀ clear table_class table_d u
                • updateField clear table_class (u, table_d)
                  = (if clear = table_class
                     then
                       if isItem u
                       then
                         giveVal
                           (MkData (Dat_class table_d) (destItem u))
                       else if isClass u
                       then
                         if destClass u dominates Dat_class table_d
                         then
                           giveVal
                             (MkData
                                 (destClass u)
                                 (Dat_item table_d))
                         else giveError downGrade
                       else giveVal (destData u)
                     else if isItem u
                     then
                       if Dat_class table_d dominates clear
                       then
                         giveVal
                           (MkData (Dat_class table_d) (destItem u))
                       else giveError underClassified
                     else giveError classChange)
updateRow     ⊢ ∀ clear table_class us r
                • updateRow clear table_class (us, r)
                  = (if ¬ us ∈ Functional
                     then giveError [ambiguousUpdate]
                     else
                       (let us'
                            = RelCombine us (R_data r)
                              ⨾ Graph
                                (updateField clear table_class)
                         in let es
                                = (us' ▷ {x|isError x})
                                  ⨾ Graph destError
                            in if es = {}
```

```
                then
                  giveVal
                    (MkRow
                        (R_exist r)
                        (R_data r ⊕ us' ⨾ Graph destVal))
                else giveError (RelList (Squash es))))
```

updateQuery

```
⊢ ∀ clear i us s ts
  • updateQuery (clear, (i, us), s, ts)
      = (let colnums
             = {n
             |∃ c
               • c ∈ visibleCols clear ts
                   ∧ CS_posn c = n}
         in if ¬ us ∈ Functional
         then (s, [ambiguousUpdate])
         else if ¬ Dom (⋃ (Ran us)) ⊆ colnums
         then (s, [noSuchColumn])
         else
          (let us' = revealRow clear ts ~ ⨾ us
           in let pr
                  = RelCombine
                      us'
                      (ListRel (TS_rows ts))
                    ⨾ Graph
                      (updateRow clear (TS_class ts))
              in let es
                     = (pr ▷ {x|isError x})
                       ⨾ Graph destError
                in if es = {}
                then
                  let rs
                        = RelList
                          (ListRel (TS_rows ts)
                              ⊕ pr
                              ⨾ Graph destVal)
                    in (changeSpec
                          i
                          (replaceRows ts rs)
                          s, [])
                else (s, Flat (RelList (Squash es)))))
```

updateStateR

```
⊢ ∀ clear ef es s
  • updateStateR (clear, (ef, es), s)
      = (if isSelect ef
        then (s, clear, destSelect ef, es)
        else if ¬ es = []
        then (s, clear, [], es)
        else
```

```
(let i = tabFromEffect ef
  in if tabExists clear i s
  then
    let ts = getTable i s
      in if ¬ clear dominates TS_class ts
      then (s, clear, [], [accessDenied])
      else
        (let (s', es')
              = (if isInsert ef
                then
                  insertQuery
                    (clear, destInsert ef, s,
                       ts)
                else if isDelete ef
                then
                  (deleteQuery
                        (clear,
                           destDelete ef, s,
                           ts), [])
                else
                  updateQuery
                    (clear, destUpdate ef, s,
                       ts))
           in (s', clear, [], es'))
  else (s, clear, [], [noSuchTable])))
```

updateState

```
⊢ ∀ clear ef es s
  • updateState (clear, (ef, es), s)
    = (let (s_r, out)
          = updateStateR
            (clear, (ef, es), repState s)
      in (absState s_r, out))
```

7 INDEX