

*Project:* DRA FRONT END FILTER PROJECT

*Title:* Specification of SSQL Semantics II

*Ref:* DS/FMU/FEF/014      *Issue: Revision : 2.6*      *Date:* 5 December 2009

*Status:* Draft      *Type:* Specification

*Keywords:*

*Author:*

<i>Name</i>	<i>Location</i>	<i>Signature</i>	<i>Date</i>
G. M. Prout	WIN01		

*Authorisation for Issue:*

<i>Name</i>	<i>Function</i>	<i>Signature</i>	<i>Date</i>
R.B. Jones	HAT Manager		

*Abstract:* The formal specification of the function *processQuery*; this completes the specifications of the main functionality of the SSQL semantics for the DRA front end filter project RSRE 1C/6130.

*Distribution:* HAT FEF File  
Simon Wiseman

## 0 DOCUMENT CONTROL

### 0.1 Contents List

<b>0</b>	<b>DOCUMENT CONTROL</b>	<b>2</b>
0.1	Contents List . . . . .	2
0.2	Document Cross References . . . . .	2
0.3	Changes History . . . . .	3
0.4	Changes Forecast . . . . .	3
<b>1</b>	<b>GENERAL</b>	<b>4</b>
1.1	Scope . . . . .	4
1.2	Introduction . . . . .	4
1.3	Terminology . . . . .	4
<b>2</b>	<b>PRELIMINARIES</b>	<b>4</b>
<b>3</b>	<b>CONSTRUCTORS ETC.</b>	<b>4</b>
<b>4</b>	<b>CONTINUATION OF TRANSLATION</b>	<b>6</b>
<b>5</b>	<b>GIVEN FUNCTIONS</b>	<b>7</b>
<b>6</b>	<b>AUXILIARY FUNCTIONS</b>	<b>13</b>
<b>7</b>	<b>SYNTACTIC CLAUSES</b>	<b>24</b>
<b>8</b>	<b>SYNTACTIC FUNCTIONS</b>	<b>27</b>
<b>9</b>	<b>AUXILIARY FUNCTIONS - CONTINUED</b>	<b>27</b>
<b>10</b>	<b>SYNTACTIC FUNCTIONS - CONTINUED</b>	<b>35</b>
<b>11</b>	<b>CLOSING DOWN</b>	<b>51</b>
<b>12</b>	<b>INDEX</b>	<b>52</b>

### 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] DS/FMU/FEF/005. *Specifications of hide and updateState*. G.M. Prout, ICL Secure Systems, WIN01.
- [4] *The Specification of Secure SQL*. Simon Wiseman, DRA, 6th July 1992.

### **0.3 Changes History**

**Issue** *Revision : 2.6 (5 December 2009)* Definition given for *gbl*.

**Issue 2.6** Removed dependency on ICL logo font

### **0.4 Changes Forecast**

None.

# 1 GENERAL

## 1.1 Scope

This document completes the formal specifications of the functionality of the SSQL semantics. Together with [2] it constitutes deliverable D3 of work package 1a, as given in section 7 of the Secure Database Technical Proposal, [1].

## 1.2 Introduction

In this document we give a formal specification for the function *processQuery* which captures the main functionality of SSQL, i.e. how the result of a query is computed from the state of the database. This is item 2 as described in the introduction to [2]. For anything that is incompletely specified in [4], we only give its signature.

## 1.3 Terminology

# 2 PRELIMINARIES

The following ProofPower instructions set up the new theory *fef014* and set the context for the proof tools.

SML

```
| open_theory "fef005";
| (force_delete_theory "fef014" handle _ => ());
| new_theory "fef014";
| push_pc "hol";
```

We provide *true* as an alias for *T* and *false* as an alias for *F*.

SML

```
| declare_alias("true", ⌈ T ⌋);
| declare_alias("false", ⌈ F ⌋);
```

# 3 CONSTRUCTORS ETC.

First, we need destructor and discriminator functions for things of type *Item* and *Val*.

HOL Constant

```
| destValuedItem      : Item → ValuedItem;
| destNullItem        : Item → NullItem
```

```
| ∀      v:ValuedItem; n:NullItem
| •      destValuedItem (ValuedItemItem v) = v
| ∧      destNullItem(NullItemItem n)     = n
```

HOL Constant

**isValuedItem** :  $Item \rightarrow Bool$ ;**isNullItem** :  $Item \rightarrow Bool$  $\forall$   $i:Item$ •  $(isValuedItem\ i = \exists v:ValuedItem \bullet i = ValuedItemItem\ v)$  $\wedge$   $(isNullItem\ i = \exists n:NullItem \bullet i = NullItemItem\ n)$ 

HOL Constant

**destBoolVal** :  $Val \rightarrow Bool$ ;**destStringVal** :  $Val \rightarrow String$ ;**destIntVal** :  $Val \rightarrow Int$ ;**destFloatVal** :  $Val \rightarrow Float$ ;**destTimeVal** :  $Val \rightarrow Time$ ;**destIntervalVal** :  $Val \rightarrow Interval$ ;**destCodeVal** :  $Val \rightarrow Code$ ;**destClassVal** :  $Val \rightarrow Class$  $\forall$   $b\ s\ i\ f\ t\ int\ c\ cl$ •  $destBoolVal\ (BoolVal\ b) = b$  $\wedge$   $destStringVal\ (StringVal\ s) = s$  $\wedge$   $destIntVal\ (IntVal\ i) = i$  $\wedge$   $destFloatVal\ (FloatVal\ f) = f$  $\wedge$   $destTimeVal\ (TimeVal\ t) = t$  $\wedge$   $destIntervalVal\ (IntervalVal\ int) = int$  $\wedge$   $destCodeVal\ (CodeVal\ c) = c$  $\wedge$   $destClassVal\ (ClassVal\ cl) = cl$

HOL Constant

---

```

isBoolVal      : Val → Bool;
isStringVal   : Val → Bool;
isIntVal      : Val → Bool;
isFloatVal    : Val → Bool;
isTimeVal     : Val → Bool;
isIntervalVal : Val → Bool;
isCodeVal     : Val → Bool;
isClassVal    : Val → Bool

```

---

```

∀      v : Val
•      (isBoolVal v      = ∃ b:Bool      • v = BoolVal b)
∧      (isStringVal v   = ∃ s:String    • v = StringVal s)
∧      (isIntVal v      = ∃ i:Int       • v = IntVal i)
∧      (isFloatVal v    = ∃ f:Float     • v = FloatVal f)
∧      (isTimeVal v     = ∃ t:Time      • v = TimeVal t)
∧      (isIntervalVal v = ∃ int:Interval • v = IntervalVal int)
∧      (isCodeVal v     = ∃ c:Code      • v = CodeVal c)
∧      (isClassVal v    = ∃ cl:Class    • v = ClassVal cl)

```

## 4 CONTINUATION OF TRANSLATION

SML

```

|declare_type_abbrev("Col",[],⌈: Ide LIST⌋);

```

SML

```

|declare_type_abbrev("Tuple",[],⌈: Col → Data + Errors⌋);

```

HOL Labelled Product

---

*GroupedResult*


---

```

G_res      : Data LIST;
G_group    : Tuple LIST

```

---

The type *Maybe* is given as a one-point type.

SML

```

|declare_type_abbrev("Maybe",[],⌈: ONE⌋);
|declare_alias("maybe",⌈One⌋);

```

SML

```
|declare_type_abbrev("MaybeResult",[],⌈: Maybe + Data LIST⌋);
```

HOL Labelled Product

*Env*

<b>E_row</b>	: <i>Tuple</i> ;
<b>E_group</b>	: <i>Tuple LIST</i>

*userName*, *userDirectory*, *userClearance* and *timeNow* are modelled as global variables.

HOL Constant

<b>userName</b>	: <i>String</i> ;
<b>userDirectory</b>	: <i>Ide LIST</i> ;
<b>userClearance</b>	: <i>Class</i> ;
<b>timeNow</b>	: <i>Time</i>

<i>true</i>
-------------

## 5 GIVEN FUNCTIONS

In this section, signatures only are given for those functions that have not been defined in the informal specification.

For convenience, we give the least upper bound and greatest lower bound over a sequence of classes.

HOL Constant

<b>lubl</b>	: <i>Class LIST</i> $\rightarrow$ <i>Class</i>
-------------	--

$\forall cl \bullet \text{lubl } cl = \text{Fold } \$\text{lub } cl \text{ lattice\_bottom}$
--

HOL Constant

<b>gbl</b>	: <i>Class LIST</i> $\rightarrow$ <i>Class</i>
------------	--

$\forall cl \bullet \text{gbl } cl = \text{Fold } \$\text{glb } cl \text{ lattice\_top}$
--

SML

```
|declare_infix (150,"dominates_w");
```

HOL Constant

<b>\$dominates_w</b>	: <i>Worth</i> $\rightarrow$ <i>Worth</i> $\rightarrow$ <i>Bool</i>
----------------------	---

<i>true</i>
-------------

SML

| *declare\_infix* (150,"lub\_w");

HOL Constant

|  $\$lub\_w : Worth \rightarrow Worth \rightarrow Worth$ | *true*

We give the least upper bound over a sequence of worths.

HOL Constant

|  $\$lub\_wl : Worth\ LIST \rightarrow Worth$ | *true*

The different ‘sorts’ of operators are modelled as strings.

HOL Constant

| **Not And Or Plus Equal** : *String*| *Not* = "*Not*"|  $\wedge$  *And* = "*And*"|  $\wedge$  *Or* = "*Or*"|  $\wedge$  *Plus* = "*Plus*"|  $\wedge$  *Equal* = "*Equal*"

SML

| *declare\_type\_abbrev*("Op",[ $\Gamma$ : *String*]);

An auxiliary function to obtain *Data* from a classification, a worth and a value.

HOL Constant

| **newData** : *Class*  $\rightarrow$  *Worth*  $\rightarrow$  *Val*  $\rightarrow$  *Data*|  $\forall c\ w\ v \bullet \text{newData } c\ w\ v =$   
|  $\text{MkData } c\ (\text{ValuedItemItem}(\text{MkValuedItem } w\ v))$ 

We require the least upper bound of clearances from a list of data together with the clearance of the user.

HOL Constant

| **lub\_data** : *Data LIST*  $\rightarrow$  *Class*|  $\forall dl \bullet \text{lub\_data } dl = (\text{userClearance } \text{lub } (\text{lubl } (\text{Map } \text{Dat\_class } dl)))$

Similarly for worths.

HOL Constant

$$\mathbf{lub\_wdata} : Data\ LIST \rightarrow Worth$$

$$\forall dl \bullet lub\_wdata\ dl = lub\_wl(Map\ (VI\_worth\ o\ destValuedItem\ o\ Dat\_item)\ dl)$$

*ExceptionData* has classification at the least upper bound of a list of data and the clearance of the user. It's value is *ExceptionVal*, with worth *worthless*.

HOL Constant

$$\mathbf{ExceptionData} : Data\ LIST \rightarrow Data$$

$$\forall dl \bullet ExceptionData\ dl = newData\ (lub\_data\ dl)\ worthless\ ExceptionVal$$

*NullData* also has classification at the least upper bound of a list of data and the clearance of the user.

HOL Constant

$$\mathbf{NullData} : Data\ LIST \rightarrow Data$$

$$\forall dl \bullet NullData\ dl = MkData\ (lub\_data\ dl)\ (NullItemItem\ null)$$

The specification in [4] of the semantics of the application of operators is incomplete. We have not modelled separately the 'isNotCleared' case from [4] because the data obtained from the application of an operator is always classified at the least upper bound of the classification of the inputs and the clearance of the user. If that least upper bound is not dominated by the user's clearance then the data will be treated in the same way as a classification result is treated in [4].

HOL Constant

$$\mathbf{applyNot} : Data\ LIST \rightarrow Data$$

$$\begin{aligned} \forall dl \bullet applyNot\ dl = & \\ & \text{if } \neg(\#dl = 1) \\ & \text{then } ExceptionData\ dl \\ & \text{else if } isNullItem\ (Dat\_item\ (Hd\ dl)) \\ & \text{then } Hd\ dl \\ & \text{else let } v = VI\_val(destValuedItem(Dat\_item\ (Hd\ dl))) \\ & \quad \text{in} \\ & \quad \text{if } isBoolVal\ v \\ & \quad \text{then } newData(lub\_data\ dl)(lub\_wdata\ dl)(BoolVal(\neg\ (destBoolVal\ v))) \\ & \quad \text{else } ExceptionData\ dl \end{aligned}$$

HOL Constant

**applyAnd** : *Data LIST* → *Data*


---

```

∀ dl • applyAnd dl =
  if ¬(#dl = 2)
  then ExceptionData dl
  else
    let d1 = Hd dl and d2 = Hd(Tl dl)
    in
      if isValuedItem (Dat_item d1)
      then
        let v1 = VI_val(destValuedItem(Dat_item d1))
        in
          if isValuedItem (Dat_item d2)
          then
            let v2 = VI_val(destValuedItem(Dat_item d2))
            in
              if ((isBoolVal v1) ∧ (isBoolVal v2))
              then newData(lub_data dl)(lub_wdata dl)
                (BoolVal((destBoolVal v1) ∧ (destBoolVal v2)))
              else if (isBoolVal v1) ∧ (destBoolVal v1 = false)
              then newData(lub_data dl)(lub_wdata dl)(BoolVal false)
              else if (isBoolVal v2) ∧ (destBoolVal v2 = false)
              then newData(lub_data dl)(lub_wdata dl)(BoolVal false)
              else ExceptionData dl
            else NullData dl
          else NullData dl
        else NullData dl
      else NullData dl

```

HOL Constant

---

**applyOr** : *Data LIST* → *Data*


---

```

∀ dl • applyOr dl =
  if ¬(#dl = 2)
  then ExceptionData dl
  else
    let d1 = Hd dl and d2 = Hd(Tl dl)
    in
      if isValuedItem (Dat_item d1)
      then
        let v1 = VI_val(destValuedItem(Dat_item d1))
        in
          if isValuedItem (Dat_item d2)
          then
            let v2 = VI_val(destValuedItem(Dat_item d2))
            in
              if ((isBoolVal v1) ∧ (isBoolVal v2))
              then newData(lub_data dl)(lub_wdata dl)
                (BoolVal((destBoolVal v1) ∨ (destBoolVal v2)))
              else if (isBoolVal v1) ∧ (destBoolVal v1 = true)
              then newData(lub_data dl)(lub_wdata dl)(BoolVal true)
              else if (isBoolVal v2) ∧ (destBoolVal v2 = true)
              then newData(lub_data dl)(lub_wdata dl)(BoolVal true)
              else ExceptionData dl
            else NullData dl
          else NullData dl
        else NullData dl
      else NullData dl

```

We give the signature for a plus operator on type *Int*.

SML

```

declare_infix (300,"intPlus");

```

HOL Constant

---

**\$intPlus** : *Int* → *Int* → *Int*


---

```

true

```

HOL Constant

**applyPlus** : *Data LIST* → *Data*


---

$\forall dl \bullet$  *applyPlus dl* =  
 if  $\neg(\#dl = 2)$   
 then *ExceptionData dl*  
 else  
 let  $d_1 = Hd\ dl$  and  $d_2 = Hd(Tl\ dl)$   
 in  
   if (*isNullItem (Dat\_item d<sub>1</sub>)*  $\vee$  *isNullItem (Dat\_item d<sub>2</sub>)*)  
   then *NullData dl*  
   else  
   let  $v_1 = VI\_val(destValuedItem(Dat\_item\ d_1))$   
   and  $v_2 = VI\_val(destValuedItem(Dat\_item\ d_2))$   
   in if (*isIntVal v<sub>1</sub>*)  $\wedge$  (*isIntVal v<sub>2</sub>*)  
   then *newData(lub\_data dl)(lub\_wdata dl)*  
       (*IntVal((destIntVal v<sub>1</sub>) intPlus (destIntVal v<sub>2</sub>))*)  
   else *ExceptionData dl*

HOL Constant

**applyEqual** : *Data LIST* → *Data*


---

$\forall dl \bullet$  *applyEqual dl* =  
 if  $\neg(\#dl = 2)$   
 then *ExceptionData dl*  
 else  
 let  $d_1 = Hd\ dl$  and  $d_2 = Hd(Tl\ dl)$   
 in  
   if (*isNullItem (Dat\_item d<sub>1</sub>)*  $\vee$  *isNullItem (Dat\_item d<sub>2</sub>)*)  
   then *NullData dl*  
   else  
   let  $v_1 = VI\_val(destValuedItem(Dat\_item\ d_1))$   
   and  $v_2 = VI\_val(destValuedItem(Dat\_item\ d_2))$   
   in *newData(lub\_data dl)(lub\_wdata dl)(BoolVal(v<sub>1</sub> = v<sub>2</sub>))*

HOL Constant

$$\mathbf{apply} : Op \rightarrow Data\ LIST \rightarrow Data$$


---


$$\begin{aligned} \forall dl \bullet \quad & apply\ Not\ dl &= applyNot\ dl \\ \wedge \quad & apply\ And\ dl &= applyAnd\ dl \\ \wedge \quad & apply\ Or\ dl &= applyOr\ dl \\ \wedge \quad & apply\ Plus\ dl &= applyPlus\ dl \\ \wedge \quad & apply\ Equal\ dl &= applyEqual\ dl \end{aligned}$$

We will use  $\hat{\ }^{\wedge}$  and *Cons*, where appropriate, for concatenation of two lists and prepending or appending an element to a list.

## 6 AUXILIARY FUNCTIONS

SML

```
declare_infix (300, "*1");
```

HOL Constant

$$\mathbf{\$*}_1 : ('x \rightarrow 'y + Errors) \rightarrow ('y \rightarrow 'z + Errors) \rightarrow ('x \rightarrow 'z + Errors)$$


---


$$\begin{aligned} \forall f\ g\ x \bullet (f\ *_1\ g)\ x = & \quad \text{if} \quad isVal\ (f\ x) \\ & \text{then} \quad g\ (destVal\ (f\ x)) \\ & \text{else} \quad giveError\ (destError\ (f\ x)) \end{aligned}$$

SML

```
declare_infix (300, "*2");
```

HOL Constant

$$\mathbf{\$*}_2 : ('x + Errors) \rightarrow ('x \rightarrow 'y + Errors) \rightarrow ('y + Errors)$$


---


$$\begin{aligned} \forall x\ f \bullet (x\ *_2\ f) = & \quad \text{if} \quad isVal\ x \\ & \text{then} \quad f\ (destVal\ x) \\ & \text{else} \quad giveError\ (destError\ x) \end{aligned}$$

SML

```
declare_infix (300, "*3");
```

HOL Constant

$$\mathbf{\$*}_3 : ('x + Errors) \rightarrow ('x \rightarrow ('y \rightarrow 'z + Errors)) \rightarrow ('y \rightarrow 'z + Errors)$$


---


$$\begin{aligned} \forall x\ f\ y \bullet (x\ *_3\ f)\ y = & \quad \text{if} \quad isVal\ x \\ & \text{then} \quad (f\ (destVal\ x))\ y \\ & \text{else} \quad giveError\ (destError\ x) \end{aligned}$$

We give further functions of this form.

SML

```
|declare_infix (300,"*_4");
```

HOL Constant

$$\$_{*}_4 : ('x \rightarrow 'y + Errors) \rightarrow ('y \rightarrow 'z) \rightarrow ('x \rightarrow 'z + Errors)$$

$$\forall f g x \bullet (f *_4 g) x = \begin{array}{ll} \text{if} & \text{isVal } (f x) \\ \text{then} & \text{giveVal}(g(\text{destVal}(f x))) \\ \text{else} & \text{giveError}(\text{destError } (f x)) \end{array}$$

SML

```
|declare_infix (300,"*_5");
```

HOL Constant

$$\$_{*}_5 : ('x + Errors) \rightarrow ('x \rightarrow 'y) \rightarrow ('y + Errors)$$

$$\forall x f \bullet (x *_5 f) = \begin{array}{ll} \text{if} & \text{isVal } x \\ \text{then} & \text{giveVal}(f (\text{destVal } x)) \\ \text{else} & \text{giveError } (\text{destError } x) \end{array}$$

SML

```
|declare_infix (300,"*_6");
```

HOL Constant

$$\$_{*}_6 : (('x + Errors) \times ('y + Errors)) \rightarrow ('x \rightarrow 'y \rightarrow 'z) \rightarrow ('z + Errors)$$

$$\forall f x y \bullet ((x,y) *_6 f) = \begin{array}{ll} \text{if} & \text{isVal } x \\ \text{then} & \text{if isVal } y \\ & \text{then giveVal}(f(\text{destVal } x)(\text{destVal } y)) \\ & \text{else giveError } (\text{destError } y) \\ \text{else} & \text{if isVal } y \\ & \text{then giveError } (\text{destError } x) \\ & \text{else giveError } ((\text{destError } x) \wedge (\text{destError } y)) \end{array}$$

SML

```
|declare_infix (300,"*_7");
```

HOL Constant

$$\$_{*}_7 : 'x \rightarrow ('x \rightarrow 'y) + Errors \rightarrow ('y + Errors)$$

$$\forall f x \bullet (x *_7 f) = \begin{array}{ll} \text{if} & \text{isVal } f \\ \text{then} & \text{giveVal}((\text{destVal } f) x) \\ \text{else} & \text{giveError } (\text{destError } f) \end{array}$$

We cannot alias all the above because resolution would be ambiguous.

SML

```

| declare_alias("*,⌈$*_1⌋");
| declare_alias("*,⌈$*_2⌋");
| declare_alias("*,⌈$*_3⌋");
| declare_alias("*,⌈$*_4⌋");
| declare_alias("*,⌈$*_6⌋");

```

We will use *Hd*, *Tl*, *Front* and *Last* for taking apart lists.

SML

```

| declare_infix (150,"list_⊔");

```

HOL Constant

$\mathbf{\$list\_⊔} : 'x \text{ LIST} \rightarrow 'x \text{ LIST} \rightarrow 'x \text{ LIST}$
$\forall l_1 \bullet (l_1 \text{ list\_⊔ } []) = l_1$
$\wedge \quad \forall l_2 x \bullet$
$\quad (l_1 \text{ list\_⊔ } (l_2 \hat{\ } [x]))$
$= \quad \text{if } x \in \text{Elems } l_1 \text{ then } l_1 \text{ list\_⊔ } l_2$
$\quad \text{else } (l_1 \text{ list\_⊔ } l_2) \hat{\ } [x]$

We will use *#* for the length of a list and *Flat* for concatenating a list of lists into a single list.

SML

```

| declare_infix (150,"&_1");

```

HOL Constant

$\mathbf{\$&_1} : ('x + \text{Errors}) \rightarrow ('x + \text{Errors}) \rightarrow ('x \text{ LIST} + \text{Errors})$
$\forall x_1 x_2 \bullet (x_1 \&_1 x_2) =$
$\text{if } \quad \text{isVal } x_1$
$\quad \text{then } \quad \text{if } \quad \text{isVal } x_2$
$\quad \quad \text{then } \quad \text{giveVal}([destVal x_1] \hat{\ } [destVal x_2])$
$\quad \quad \text{else } \text{giveError } (destError x_2)$
$\text{else } \quad \text{if } \quad \text{isVal } x_2$
$\quad \text{then } \quad \text{giveError } (destError x_1)$
$\quad \text{else } \quad \text{giveError}(destError x_1 \hat{\ } destError x_2)$

SML

```

| declare_infix (150,"&_2");

```

HOL Constant

$$\mathcal{S}\&_2 : ('x + Errors) \rightarrow ('x LIST + Errors) \rightarrow ('x LIST + Errors)$$


---


$$\begin{aligned} &\forall x xl \bullet (x \&_2 xl) = \\ &\text{if } \quad \text{isVal } x \\ &\quad \text{then } \quad \text{if } \quad \text{isVal } xl \\ &\quad \quad \text{then } \quad \text{giveVal}(\text{Cons}(\text{destVal } x) (\text{destVal } xl)) \\ &\quad \quad \text{else } \quad \text{giveError } (\text{destError } xl) \\ &\text{else } \quad \text{if } \quad \text{isVal } xl \\ &\quad \text{then } \quad \text{giveError } (\text{destError } x) \\ &\quad \text{else } \quad \text{giveError}(\text{destError } x \hat{\wedge} \text{destError } xl) \end{aligned}$$

SML

```
|declare_infix (150,"&3");
```

HOL Constant

$$\mathcal{S}\&_3 : ('x LIST + Errors) \rightarrow ('x + Errors) \rightarrow ('x LIST + Errors)$$


---


$$\begin{aligned} &\forall xl x \bullet (xl \&_3 x) = \\ &\text{if } \quad \text{isVal } x \\ &\quad \text{then } \quad \text{if } \quad \text{isVal } xl \\ &\quad \quad \text{then } \quad \text{giveVal}((\text{destVal } xl) \hat{\wedge} [\text{destVal } x]) \\ &\quad \quad \text{else } \quad \text{giveError } (\text{destError } xl) \\ &\text{else } \quad \text{if } \quad \text{isVal } xl \\ &\quad \text{then } \quad \text{giveError } (\text{destError } x) \\ &\quad \text{else } \quad \text{giveError}(\text{destError } xl \hat{\wedge} \text{destError } x) \end{aligned}$$

SML

```
|declare_infix (150,"&4");
```

HOL Constant

$$\mathcal{S}\&_4 : ('x LIST + Errors) \rightarrow ('x LIST + Errors) \rightarrow ('x LIST + Errors)$$


---


$$\begin{aligned} &\forall xl_1 xl_2 \bullet (xl_1 \&_4 xl_2) = \\ &\text{if } \quad \text{isVal } xl_1 \\ &\quad \text{then } \quad \text{if } \quad \text{isVal } xl_2 \\ &\quad \quad \text{then } \quad \text{giveVal}((\text{destVal } xl_1) \hat{\wedge} (\text{destVal } xl_2)) \\ &\quad \quad \text{else } \quad \text{giveError } (\text{destError } xl_2) \\ &\text{else } \quad \text{if } \quad \text{isVal } xl_2 \\ &\quad \text{then } \quad \text{giveError } (\text{destError } xl_1) \\ &\quad \text{else } \quad \text{giveError}((\text{destError } xl_1) \hat{\wedge} (\text{destError } xl_2)) \end{aligned}$$

SML

```

declare_infix(150,"&");
declare_alias("&","⌈$&_1⌋");
declare_alias("&","⌈$&_2⌋");
declare_alias("&","⌈$&_3⌋");
declare_alias("&","⌈$&_4⌋");

```

HOL Constant

$$\mathbf{seq} : Num \rightarrow 'x \rightarrow 'x \text{ LIST}$$

$$\forall n \ x \bullet \text{seq } n \ x = \begin{array}{l} \text{if } n = 0 \text{ then } [] \\ \text{else } \text{Cons } x \ (\text{seq } (n - 1) \ x) \end{array}$$

HOL Constant

$$\mathbf{enseq} : (Num \rightarrow 'x + Errors) \rightarrow 'x \text{ LIST}$$

$$\forall f \bullet \text{enseq } f = \text{RelList } \{(i,j) \mid 1 \leq i \wedge (\forall k \bullet 1 \leq k \wedge k \leq i \Rightarrow \text{isVal } (f \ k)) \wedge j = \text{destVal}(f \ i)\}$$

HOL Constant

$$\mathbf{promote} : ('x \rightarrow 'y + Errors) \rightarrow ('x \text{ LIST} \rightarrow 'y \text{ LIST} + Errors)$$

$$\forall f \ s \bullet \text{promote } f \ s = \begin{array}{l} \text{if } s = [] \text{ then } \text{giveVal} [] \\ \text{else } (f \ (\text{Hd } s) \ \& \ \text{promote } f \ (\text{Tl } s)) \end{array}$$

Note : We will use the standard constant *Map* in the cases where *promote* has been used in the informal specification with functions that do not return sum types.

HOL Constant

$$\mathbf{find} : 'x \text{ LIST} \rightarrow 'x \rightarrow Num + Errors$$

$$\forall s \ e \bullet \text{find } s \ e = \begin{array}{l} \text{if } s = [] \text{ then } \text{giveError}(\text{seq } 1 \ \text{error}) \\ \text{else if } (\text{Hd } s) = e \text{ then } \text{giveVal } 1 \\ \text{else if } \text{isVal}(\text{find } (\text{Tl } s) \ e) \text{ then} \\ \quad \text{giveVal}(1 + \text{destVal}(\text{find } (\text{Tl } s) \ e)) \\ \text{else } (\text{find } (\text{Tl } s) \ e) \end{array}$$

The function *distinct* has been specified in terms of *Ellems*, which takes a list and returns the set of elements in the list.

HOL Constant

**distinct** : 'x LIST → 'x LIST

---


$$\text{distinct } [] = []$$

$$\wedge \forall x l \bullet \text{distinct } (l \hat{\ } [x]) =$$

$$\quad \text{if } (x \in \text{Elems } l) \text{ then } \text{distinct } l$$

$$\quad \text{else } (\text{distinct } l) \hat{\ } [x]$$

HOL Constant

**noErrors** : Errors

---


$$\text{noErrors} = []$$

HOL Constant

**one\_col** : Data LIST LIST → Data LIST + Errors

---


$$\text{one\_col} = \text{promote } (\lambda ds \bullet$$

$$\quad \text{if } \#ds = 1 \text{ then } \text{giveVal}(\text{Hd } ds)$$

$$\quad \text{else } \text{giveError}(\text{seq } 1 \text{ tooWide}))$$

HOL Constant

**one\_result** : Data LIST → Data + Errors

---


$$\forall ds \bullet \text{one\_result } ds =$$

$$\quad \text{if } \#ds = 1 \text{ then } \text{giveVal}(\text{Hd } ds)$$

$$\quad \text{else } \text{giveError}(\text{seq } 1 \text{ tooTall})$$

HOL Constant

**make\_data** : Val → Data

---


$$\forall v \bullet \text{make\_data } v = \text{newData } \text{userClearance } \text{worthless } v$$

HOL Constant

**fillTab** : Tab → Tab

---


$$\forall i \bullet \text{fillTab } i = \text{if } \#i = 1 \text{ then } \text{userDirectory } \hat{\ } i \text{ else } i$$

HOL Constant

**fillCol** : Tab → Tab

---


$$\forall i \bullet \text{fillCol } i = \text{if } \#i = 2 \text{ then } \text{userDirectory } \hat{\ } i \text{ else } i$$

We need access functions for obtaining directories from states, tables from directories, column positions from tables and data from column numbers.

HOL Constant

$$\mathbf{getDir} : State \rightarrow Tab \rightarrow Directory + Errors$$

$$\begin{aligned} \forall st\ tab \bullet & \mathit{getDir}\ st\ tab = \\ & \text{if } tab \in \mathit{Dom}\ (\mathit{repState}\ st) \text{ then } \mathit{giveVal}(\mathit{repState}\ st\ @\ tab) \\ & \text{else } \mathit{giveError}\ (\mathit{seq}\ 1\ \mathit{noSuchDirectory}) \end{aligned}$$

HOL Constant

$$\mathbf{getTab} : Directory \rightarrow Ide \rightarrow TableSpec + Errors$$

$$\begin{aligned} \forall dir\ i \bullet & \mathit{getTab}\ dir\ i = \\ & \text{if } i \in \mathit{Dom}\ (\mathit{Dir\_tables}\ dir) \text{ then } \mathit{giveVal}(\mathit{Dir\_tables}\ dir\ @\ i) \\ & \text{else } \mathit{giveError}\ (\mathit{seq}\ 1\ \mathit{noSuchTable}) \end{aligned}$$

HOL Constant

$$\mathbf{getColPosn} : TableSpec \rightarrow Ide \rightarrow Num + Errors$$

$$\begin{aligned} \forall ts\ i \bullet & \mathit{getColPosn}\ ts\ i = \\ & \text{if } i \in \{name \mid \exists c \bullet c \in \mathit{TS\_colspecs}\ ts \wedge \mathit{CS\_ide}\ c = name\} \\ & \text{then } \mathit{giveVal}(\epsilon\ n \bullet \exists c \bullet c \in \mathit{TS\_colspecs}\ ts \wedge \mathit{CS\_ide}\ c = i \wedge \mathit{CS\_posn}\ c = n) \\ & \text{else } \mathit{giveError}\ (\mathit{seq}\ 1\ \mathit{noSuchColumn}) \end{aligned}$$

HOL Constant

$$\mathbf{getData} : Row \rightarrow Num \rightarrow Data + Errors$$

$$\begin{aligned} \forall r\ n \bullet & \mathit{getData}\ r\ n = \\ & \text{if } n \in \mathit{Dom}\ (\mathit{R\_data}\ r) \text{ then } \mathit{giveVal}((\mathit{R\_data}\ r)\ @\ n) \\ & \text{else } \mathit{giveError}\ (\mathit{seq}\ 1\ \mathit{noSuchColumn}) \end{aligned}$$

Now we define *lookup*. No security checks are made since the user is looking at the hidden state of the database.

HOL Constant

$$\mathbf{lookup} : State \rightarrow Tab \rightarrow TableSpec + Errors$$

$$\begin{aligned} \forall st\ tab \bullet & \mathit{lookup}\ st\ tab = \\ & \text{let } dir = \mathit{getDir}\ st\ (\mathit{Front}\ tab) \\ & \text{in } \quad \text{if } \mathit{isVal}\ dir \text{ then } \mathit{getTab}\ (\mathit{destVal}\ dir)\ (\mathit{Last}\ tab) \\ & \quad \text{else } \mathit{giveError}(\mathit{destError}\ dir) \end{aligned}$$

We supply a boolean *isCleared* which carries out the same function as *isData* applied to a *Result* in the informal specification.

HOL Constant

$$\mathbf{isCleared} : Data \rightarrow Bool$$

$$\forall d : Data \bullet isCleared\ d = (userClearance\ dominates\ (Dat\_class\ d))$$

We also supply a boolean *isNotCleared* which carries out the same function as *isClass* applied to a *Result* in the informal specification.

HOL Constant

$$\mathbf{isNotCleared} : Data \rightarrow Bool$$

$$\forall d : Data \bullet isNotCleared\ d = \neg (isCleared\ d)$$

HOL Constant

$$\mathbf{checkComplete} : Data\ LIST \rightarrow Bool$$

$$checkComplete\ [] = true$$

$$\wedge \forall d\ tl \bullet checkComplete\ (Cons\ d\ tl) =$$

$$\quad \text{if } isNotCleared\ d \text{ then } false$$

$$\quad \text{else } checkComplete\ tl$$

HOL Constant

$$\mathbf{check\_where\_complete} : Bool \rightarrow Bool \rightarrow Data \rightarrow Data + Errors$$

$$\forall comp\ b\ d \bullet check\_where\_complete\ comp\ b\ d =$$

$$\quad \text{if } comp \text{ then } giveVal\ d$$

$$\quad \text{else if } isNotCleared\ d \text{ then } giveError(seq\ 1\ notCleared)$$

$$\quad \text{else if } isNullItem\ (Dat\_item\ d) \text{ then } giveVal\ d$$

$$\quad \text{else let } v = VI\_val(destValuedItem(Dat\_item\ d))$$

$$\quad \text{in}$$

$$\quad \text{if } isBoolVal\ v \text{ then}$$

$$\quad \quad \text{if } (destBoolVal\ v) = b \text{ then } giveError(seq\ 1\ notCleared)$$

$$\quad \quad \text{else } giveVal\ d$$

$$\quad \text{else } giveError(seq\ 1\ wrongType)$$

HOL Constant

**resultBool** :  $Data \rightarrow Bool + Errors$ 


---


$$\forall d : Data \bullet \text{resultBool } d =$$

if  $\text{isNotCleared } d$  then giveVal false  
else if  $\text{isNullItem } (Dat\_item\ d)$  then giveVal false  
else let  $v = VI\_val(\text{destValuedItem}(Dat\_item\ d))$   
in  
if  $\text{isBoolVal } v$  then giveVal( $\text{destBoolVal } v$ )  
else giveError (seq 1 wrongType)

We give a function *take\_data* which returns the error *notCleared* where appropriate.

HOL Constant

**take\_data** :  $Data \rightarrow Data + Errors$ 


---


$$\forall d \bullet \text{take\_data } d = \text{if } \text{isCleared } d \text{ then giveVal } d$$

*else giveError(seq 1 notCleared)*

HOL Constant

**resultClass** :  $Data \rightarrow Class + Errors$ 


---


$$\forall d : Data \bullet \text{resultClass } d =$$

let  $vi\ i =$  if  $\text{isValuedItem } i$   
then giveVal ( $\text{destValuedItem } i$ )  
else giveError(seq 1 nullValue)  
and  $valClass\ v =$  if  $\text{isClassVal } v$   
then giveVal ( $\text{destClassVal } v$ )  
else giveError(seq 1 wrongType)  
in  
 $(((((\text{take\_data } d) *_{\text{5}} Dat\_item) * vi) *_{\text{5}} VI\_val) * valClass$

HOL Constant

**extract** :  $Bool\ LIST \rightarrow 'x\ LIST \rightarrow 'x\ LIST$ 


---


$$\forall bs\ s \bullet \text{extract } bs\ s =$$

if  $bs = []$  then  $s$   
else if  $s = []$  then  $s$   
else if  $Hd\ bs = true$  then  $Cons\ (Hd\ s)\ (\text{extract } (Tl\ bs)\ (Tl\ s))$   
else  $\text{extract } (Tl\ bs)\ (Tl\ s)$

HOL Constant

$$\mathbf{emptyTuple} : Col \rightarrow Data + Errors$$

$$\forall c \bullet \mathit{emptyTuple} \ c = \mathit{giveError} \ (\mathit{seq} \ 1 \ \mathit{noSuchColumn})$$

HOL Constant

$$\mathbf{emptyEnv} : Env$$

$$\mathit{emptyEnv} = \mathit{MkEnv} \ \mathit{emptyTuple} \ (\mathit{seq} \ 1 \ \mathit{emptyTuple})$$

We define the auxiliary functions for *join* separately.

SML

```
|declare_infix(150,"star1");
```

HOL Constant

$$\mathbf{\$star1} : Tuple \rightarrow Tuple \rightarrow Tuple$$

$$\begin{aligned} \forall a \ b \ c \bullet & \ (a \ \mathit{star1} \ b) \ c = \\ & \ \mathit{if} \ a \ c = \mathit{giveError}(\mathit{seq} \ 1 \ \mathit{noSuchColumn}) \ \mathit{then} \ b \ c \\ & \ \mathit{else} \ \mathit{if} \ b \ c = \mathit{giveError}(\mathit{seq} \ 1 \ \mathit{noSuchColumn}) \ \mathit{then} \ a \ c \\ & \ \mathit{else} \ \mathit{if} \ (\mathit{isVal} \ (a \ c) \wedge \mathit{isVal} \ (b \ c)) \ \mathit{then} \ \mathit{giveError}(\mathit{seq} \ 1 \ \mathit{ambiguousColumn}) \\ & \ \mathit{else} \ \mathit{if} \ (\mathit{isError} \ (a \ c) \wedge \mathit{isError} \ (b \ c)) \ \mathit{then} \\ & \ \qquad \mathit{giveError}(\mathit{destError}(a \ c) \ \mathit{list\_}\cup \ \mathit{destError}(b \ c)) \\ & \ \mathit{else} \ \mathit{if} \ \mathit{isError} \ (a \ c) \ \mathit{then} \ a \ c \\ & \ \mathit{else} \ b \ c \end{aligned}$$

SML

```
|declare_infix(150,"star2");
```

HOL Constant

$$\mathbf{\$star2} : Tuple \rightarrow Tuple \ LIST \rightarrow Tuple \ LIST$$

$$\begin{aligned} \forall a \ b \bullet & \ (a \ \mathit{star2} \ b) = \\ & \ \mathit{if} \ b = [] \ \mathit{then} \ [] \\ & \ \mathit{else} \ \mathit{Cons} \ (a \ \mathit{star1} \ (\mathit{Hd} \ b)) \ (a \ \mathit{star2} \ (\mathit{Tl} \ b)) \end{aligned}$$

SML

```
|declare_infix(150,"star3");
```

HOL Constant

$$\mathbf{\$star3} : \text{Tuple LIST} \rightarrow \text{Tuple LIST} \rightarrow \text{Tuple LIST}$$

$$\begin{aligned} \forall a b \bullet (a \text{ star3 } b) = \\ \text{if } a = [] \text{ then } [] \\ \text{else } ((\text{Hd } a) \text{ star2 } b) \wedge ((\text{Tl } a) \text{ star3 } b) \end{aligned}$$

HOL Constant

$$\mathbf{jay} : \text{Tuple LIST} \rightarrow \text{Tuple LIST LIST} \rightarrow \text{Tuple LIST}$$

$$\begin{aligned} \forall t s \bullet \text{jay } t s = \\ \text{if } s = [] \text{ then } t \\ \text{else } \text{jay } (t \text{ star3 } (\text{Hd } s))(\text{Tl } s) \end{aligned}$$
Finally the *join* function

HOL Constant

$$\mathbf{join} : \text{Tuple LIST LIST} \rightarrow \text{Tuple LIST}$$

$$\text{join} = \text{jay } []$$

SML

```
declare_infix(150,"starstar1");
```

HOL Constant

$$\mathbf{\$starstar1} : \text{Tuple} \rightarrow \text{Tuple} \rightarrow \text{Tuple}$$

$$\begin{aligned} \forall a b c \bullet (a \text{ starstar1 } b) c = \\ \text{if } (b c) = \text{giveError}(\text{seq } 1 \text{ noSuchColumn}) \text{ then } a c \\ \text{else } b c \end{aligned}$$

SML

```
declare_infix(150,"starstar2");
```

HOL Constant

$$\mathbf{\$starstar2} : \text{Tuple} \rightarrow \text{Tuple LIST} \rightarrow \text{Tuple LIST}$$

$$\begin{aligned} \forall a b \bullet (a \text{ starstar2 } b) = \\ \text{if } b = [] \text{ then } [] \\ \text{else } (\text{Cons } (a \text{ starstar1 } (\text{Hd } b)) (a \text{ starstar2 } (\text{Tl } b))) \end{aligned}$$

SML

```
|declare_infix(150,"**");  
|declare_alias("**",⌈$starstar1⌋);  
|declare_alias("**",⌈$starstar2⌋);
```

The function *visibleRows* is unnecessary since *processQuery* only applies to the hidden state of the database.

Before we can proceed, we need to specify some of the Syntactic functions.

## 7 SYNTACTIC CLAUSES

We first define the different ‘sorts’ of SSQL as new types.

SML

```
|new_type("Value",0);
```

SML

```
|new_type("Col_spec",0);
```

SML

```
|new_type("Table_spec",0);
```

SML

```
|new_type("Col_name",0);
```

SML

```
|new_type("From_spec",0);
```

SML

```
|new_type("Set_clause",0);
```

SML

```
|new_type("Tuple_list",0);
```

SML

```
|new_type("Insert_list",0);
```

SML

```
|new_type("Classified_value",0);
```

SML

```
|new_type("Select_list",0);
```

SML

```
|new_type("Query",0);
```

We have only specified the signatures of the syntactic clauses, as in the informal specification. This does not provide enough information to reason about them. Ideally, we would expect to define the syntactic clauses using a recursive type definition package. Since this is not yet available in **ProofPower**, we could use axioms to specify the kind of properties, e.g. injective, that these things should have.

HOL Constant

```

denote_null : Value;
denote_void : Value;
denote_true : Value;
denote_false : Value;
denote_integer : Int → Value;
denote_string : String → Value;
denote_float : Float → Value;
denote_time : Time → Value;
denote_interval : Interval → Value;
denote_class : Class → Value;
denote_code : Code → Value

```

---

*true*

HOL Constant

```

monop : Op → Value → Value;
binop : Op → (Value × Value) → Value;
set_func_all : Op → Value → Value;
count_all : Value;
all_binop : Op → Value → Tuple_list → Value;
contents : Col_spec → Value;
classification : Col_spec → Value

```

---

*true*

HOL Constant

```

user : Value;
clearance : Value;
current_time : Value

```

---

*true*

HOL Constant

---

```

denote_col_spec : Ide LIST → Col_spec;
denote_table_spec : Ide LIST → Table_spec;
denote_col_name : Ide → Col_name

```

---

*true*

HOL Constant

---

```

from : Table_spec → From_spec;
correlate_from : Ide → Table_spec → From_spec

```

---

*true*

HOL Constant

---

```

set_value : Col_name → Value → Set_clause;
set_class : Col_name → Value → Set_clause;
set_class_and_value : Col_name → Value → Value → Set_clause

```

---

*true*

HOL Constant

---

```

all_tuples : Select_list → From_spec LIST → Value →
               Col_spec LIST → Value → Tuple_list;
distinct_tuples : Select_list → From_spec LIST → Value →
                   Col_spec LIST → Value → Tuple_list;
evaluate : Select_list → From_spec LIST → Value →
            Col_spec LIST → Value → Tuple_list

```

---

*true*

HOL Constant

---

```

insert_tuples : Tuple_list → Insert_list;
tuple : Classified_value LIST → Insert_list;
union : Insert_list → Insert_list → Insert_list

```

---

*true*

HOL Constant

---

```

classify : Value → Value → Classified_value;
classify_default : Value → Classified_value

```

---

*true*

HOL Constant

---

```

all_columns : Select_list;
select_values : Value LIST → Select_list

```

---

*true*

HOL Constant

---

```

insert : Table_spec → Col_name LIST → Insert_list → Query;
delete : From_spec → Value → Query;
update : From_spec → Set_clause LIST → Value → Col_spec LIST
          → Value → Query;
select : Tuple_list → Query

```

---

*true*

## 8 SYNTACTIC FUNCTIONS

We define enough of the syntactic functions to be able to complete the definitions of the auxiliary functions.

HOL Constant

---

```

Col_spec : Col_spec → Col

```

---

$\forall il \bullet \text{Col\_spec } (\text{denote\_col\_spec } il) = \text{fillCol } il$

HOL Constant

---

```

Table_spec : Table_spec → Tab

```

---

$\forall il \bullet \text{Table\_spec } (\text{denote\_table\_spec } il) = \text{fillTab } il$

## 9 AUXILIARY FUNCTIONS - CONTINUED

We continue with the translation of the auxiliary functions from section 6. Note that we do not check the existence class of rows or columns since we are viewing the hidden state of the database.

HOL Constant

---

**projectTuples** : *State*  $\rightarrow$  *Table\_spec*  $\rightarrow$  *Tab*  $\rightarrow$  *Tuple LIST* + *Errors*


---

```

 $\forall$  st ts cn • projectTuples st ts cn =
  let table = Table_spec ts
  in
  let spec = lookup st table
  in
  if isError spec then giveError(destError spec)
  else
  let   coln c =      if Front c = table then getColPosn (destVal spec) (Last c)
                    else if Front c = cn then getColPosn (destVal spec) (Last c)
                    else giveError (seq 1 noSuchColumn)
  in
  let   entuple r = giveVal( $\lambda c$  • (coln c) * (getData r))
  in   (promote entuple)(TS_rows (destVal spec))

```

The function *changeSpec* is defined in [3] upon the representation state.

SML

```
|declare_infix (150,"&5");
```

HOL Constant

---

 $\&5$  : (*Ide*  $\rightarrow$  *Update* + *Errors*)  $\rightarrow$  (*Ide*  $\rightarrow$  *Update* + *Errors*)  
 $\rightarrow$  (*Ide*  $\rightarrow$  *Update* + *Errors*)

---

```

 $\forall$  f g c • (f &5 g) c =
  if (isError (f c)  $\wedge$  destError(f c) = seq 1 error)
     $\vee$  (isError (g c)  $\wedge$  destError(g c) = seq 1 error)
  then f c
  else if isClass (destVal(f c))  $\wedge$  isItem (destVal(g c))
  then giveVal(DataUpdate(MkData(destClass (destVal(f c)))(destItem(destVal(g c)))))
  else if isClass (destVal(g c))  $\wedge$  isItem (destVal(f c))
  then giveVal(DataUpdate(MkData(destClass (destVal(g c)))(destItem(destVal(f c)))))
  else giveError(seq 1 ambiguousUpdate)

```

SML

```
|declare_infix (150,"&6");
```

HOL Constant

$$\begin{aligned} \&_6 : (Ide \rightarrow Update + Errors) \rightarrow (Ide \rightarrow Update + Errors)LIST \\ &\rightarrow (Ide \rightarrow Update + Errors) \end{aligned}$$

$$\forall f s \bullet (f \&_6 s) = \begin{cases} \text{if } s = [] \text{ then } f \\ \text{else } (f \&_5 (Hd s)) \&_6 (Tl s) \end{cases}$$

SML

```
declare_alias("&", "⌈$&_5⌋");
```

```
declare_alias("&", "⌈$&_6⌋");
```

We define *auxapply* as an auxiliary function to *engroup*.

HOL Constant

$$\mathbf{auxapply} : Col\ LIST \rightarrow Tuple\ LIST \rightarrow Env\ LIST \rightarrow Env\ LIST$$

$$\begin{aligned} \forall cs\ ts\ es \bullet \mathbf{auxapply}\ cs\ ts\ es = \\ \text{let add}\ cs\ t\ e = \\ \text{if } (promote\ t)\ cs = (promote\ (E\_row\ e))\ cs \text{ then } MkEnv(E\_row\ e)((E\_group\ e) \wedge [t]) \\ \text{else } e \\ \text{in} \\ \text{let adds}\ cs\ t = Map(\text{add}\ cs\ t) \\ \text{in} \\ \text{if } ts = [] \text{ then } es \\ \text{else } \mathbf{auxapply}\ cs\ (Tl\ ts)\ (\text{adds}\ cs\ (Hd\ ts)\ es) \end{aligned}$$

HOL Constant

$$\mathbf{engroup} : Col\_spec\ LIST \rightarrow Tuple\ LIST \rightarrow Env\ LIST$$

$$\begin{aligned} \forall g\ ts \bullet \mathbf{engroup}\ g\ ts = \mathbf{auxapply} \\ (Map(\lambda c \bullet Col\_spec\ c)\ g) \\ ts \\ (Map(\lambda t \bullet MkEnv\ t\ [])\ ts) \end{aligned}$$

The definition of *eliminate* is given locally in the functions *Tuple\_list<sub>p</sub>* and *processUpdate* in section 10.

HOL Constant

$$\mathbf{colsInGroup} : TableSpec \rightarrow Num \rightarrow Bool\ LIST$$

$$true$$

HOL Constant

$$\mathbf{andBools} : Bool\ LIST \rightarrow Bool$$

$$\forall bs \bullet \mathbf{andBools}\ bs = \begin{array}{l} \text{if } \#bs = 0 \text{ then true} \\ \text{else } (Hd\ bs) \wedge \mathbf{andBools}(Tl\ bs) \end{array}$$

The function *fits* is no longer relevant because of the simplifications made to the integer and floating data types.

We require access functions for retrieving *ColSpec* from a table given the column number or the column name, and *ColCon* from a table given the column group number. We use the names *colposns*, *colspecs* and *cons* since these are the names of the equivalent components of *TableSpec* in the informal specification.

HOL Constant

$$\mathbf{colposns} : TableSpec \rightarrow Num \rightarrow ColSpec + Errors$$

$$\forall t\ n \bullet \mathbf{colposns}\ t\ n = \begin{array}{l} \text{if } \exists_1\ c \bullet c \in TS\_colspecs\ t \wedge CS\_posn\ c = n \\ \text{then } giveVal\ (\epsilon\ c \bullet c \in TS\_colspecs\ t \wedge CS\_posn\ c = n) \\ \text{else } giveError(seq\ 1\ noSuchColumn) \end{array}$$

HOL Constant

$$\mathbf{colspecs} : TableSpec \rightarrow Ide \rightarrow ColSpec + Errors$$

$$\forall t\ i \bullet \mathbf{colspecs}\ t\ i = \begin{array}{l} \text{if } \exists_1\ c \bullet c \in TS\_colspecs\ t \wedge CS\_ide\ c = i \\ \text{then } giveVal\ (\epsilon\ c \bullet c \in TS\_colspecs\ t \wedge CS\_ide\ c = i) \\ \text{else } giveError(seq\ 1\ noSuchColumn) \end{array}$$

HOL Constant

$$\mathbf{cons} : TableSpec \rightarrow Num \rightarrow ColCon + Errors$$

$$\forall t\ n \bullet \mathbf{cons}\ t\ n = \begin{array}{l} \text{if } \exists_1\ cc \bullet (n,cc) \in TS\_cons\ t \\ \text{then } giveVal\ ((TS\_cons\ t) @ n) \\ \text{else } giveError(seq\ 1\ error) \end{array}$$

We define an auxiliary function for ‘anding’ something of type *Bool* + *Errors* with something of type *Bool*.

SML

```
declare_infix (40, "andb");
```

HOL Constant

$$\mathbf{\$andb} : (Bool + Errors) \rightarrow Bool \rightarrow (Bool + Errors)$$

$$\forall be\ b \bullet (be\ \mathbf{\$andb}\ b) = \begin{array}{l} \text{if } isVal\ be \\ \text{then } giveVal(destVal\ be \wedge b) \\ \text{else } giveError(destError\ be) \end{array}$$

HOL Constant

$$\mathbf{checkGroup} : TableSpec \rightarrow (ColCon \rightarrow Bool) \rightarrow (TableSpec \rightarrow Num \rightarrow Bool) \rightarrow Bool$$

$$\forall ts \ cond \ check \bullet \ checkGroup \ ts \ cond \ check = \\ \quad \mathit{andBools} (\mathit{enseq}(\lambda n \bullet ((\mathit{cons} \ ts \ n) *_{\mathcal{S}} \ cond) \ \mathit{andb} \ (check \ ts \ n)))$$

We first give a function which returns a list of data given a row.

HOL Constant

$$\mathbf{dataList} : Row \rightarrow Data \ LIST$$

$$\forall r \bullet \ dataList \ r = \mathit{RelList}(\mathit{Squash}(\mathit{R\_data} \ r))$$

HOL Constant

$$\mathbf{checkUniqueness} : TableSpec \rightarrow Num \rightarrow Bool$$

$$\forall ts \ n \bullet \ checkUniqueness \ ts \ n = \\ \quad \mathit{let} \ key = \\ \quad \quad \mathit{let} \ ii = \mathit{dataList} \ \mathcal{S} \ \mathit{extract}(\mathit{colsInGroup} \ ts \ n) \ \mathcal{S} \ \mathit{Map} \ \mathit{Dat\_item} \\ \quad \quad \mathit{in} \\ \quad \quad \mathit{TS\_rows} \ \mathcal{S} \ \mathit{Map} \ ii \\ \quad \quad \mathit{in} \\ \quad \quad \mathit{key} \ ts = \mathit{distinct} \ (key \ ts)$$

We define *test* as an auxiliary function to *checkUniform*.

HOL Constant

$$\mathbf{test} : Class \ LIST \ LIST \rightarrow Bool$$

$$\forall css \bullet \ test \ css = \quad \mathit{if} \ \#css = 0 \ \mathit{then} \ \mathit{true} \\ \quad \quad \mathit{else} \ \mathit{if} \ \#(\mathit{distinct} \ (\mathit{Hd} \ css)) = 1 \ \mathit{then} \ \mathit{test}(\mathit{Tl} \ css) \\ \quad \quad \mathit{else} \ \mathit{false}$$

HOL Constant

$$\mathbf{checkUniform} : TableSpec \rightarrow Num \rightarrow Bool$$

$$\forall ts \ n \bullet \ checkUniform \ ts \ n = \\ \quad \mathit{let} \ classes = \\ \quad \quad \mathit{let} \ cc = \mathit{dataList} \ \mathcal{S} \ \mathit{extract}(\mathit{colsInGroup} \ ts \ n) \ \mathcal{S} \ \mathit{Map} \ \mathit{Dat\_class} \\ \quad \quad \mathit{in} \\ \quad \quad \mathit{TS\_rows} \ \mathcal{S} \ \mathit{Map} \ cc \\ \quad \quad \mathit{in} \ \mathit{test} \ (classes \ ts)$$

We give a version of *seq* that scoops up errors.

HOL Constant

$$\mathbf{seqErr} : Num \rightarrow ('x + Errors) \rightarrow ('x LIST + Errors)$$

$$\begin{aligned} \forall n xe \bullet seqErr n xe = & \quad \text{if } isVal \text{ } xe \\ & \text{then } giveVal(seq n (destVal xe)) \\ & \text{else } giveError(destError xe) \end{aligned}$$

HOL Constant

$$\mathbf{checkIntegrity} : TableSpec \rightarrow (TableSpec \rightarrow Num \rightarrow Bool + Errors) \rightarrow Bool$$

$$\begin{aligned} \forall ts check \bullet checkIntegrity ts check = \\ \text{andBools } (enseq(\lambda n \bullet check ts n)) \end{aligned}$$

We define *inRange* as an auxiliary function to *checkUniform*.

HOL Constant

$$\mathbf{inRange} : Class LIST \rightarrow ColSpec LIST \rightarrow Bool$$

$$\begin{aligned} \forall cs css \bullet inRange cs css = \\ \text{let } inRan \text{ } cl \text{ } c = ((cl \text{ dominates } (CS\_min \text{ } c)) \wedge ((CS\_max \text{ } c) \text{ dominates } cl)) \\ \text{in} \\ \text{(if } \#cs = 0 \vee \#css = 0 \\ \text{then } true \\ \text{else } inRan (Hd cs) (Hd css) \wedge inRange (Tl cs) (Tl css)) \end{aligned}$$

We define *elem* in terms of *Nth*.

HOL Constant

$$\mathbf{elem} : Num \rightarrow 'a LIST \rightarrow 'a$$

$$\forall n xl \bullet elem n xl = Nth xl n$$

HOL Constant

$$\mathbf{checkFieldClasses} : TableSpec \rightarrow Num \rightarrow Bool + Errors$$

$$\begin{aligned} \forall ts n \bullet checkFieldClasses ts n = \\ \text{let } classes = (TS\_rows \text{ } \text{Map}(dataList \text{ } (elem n) \text{ } Dat\_class)) \text{ } ts \\ \text{and } cs = colposns ts n \\ \text{in } (seqErr n cs) *_5 (inRange classes) \end{aligned}$$

We define *rightType* as an auxiliary function to *CheckType*. First we need constructors for type *Type*.

HOL Constant

	<b>monoleanType</b>	: <i>Type</i> ;
	<b>booleanType</b>	: <i>Type</i> ;
	<b>charsType</b>	: <i>Type</i> ;
	<b>integerType</b>	: <i>Type</i> ;
	<b>floatingType</b>	: <i>Type</i> ;
	<b>timeType</b>	: <i>Type</i> ;
	<b>intervalType</b>	: <i>Type</i> ;
	<b>classType</b>	: <i>Type</i>
<hr/>		
	<i>monoleanType</i>	= <i>InL monolean</i>
∧	<i>booleanType</i>	= <i>InR (InL boolean)</i>
∧	<i>charsType</i>	= <i>InR (InR (InL chars))</i>
∧	<i>integerType</i>	= <i>InR (InR (InR (InL integer)))</i>
∧	<i>floatingType</i>	= <i>InR (InR (InR (InR (InL floating))))</i>
∧	<i>timeType</i>	= <i>InR (InR (InR (InR (InR (InL time))))</i>
∧	<i>intervalType</i>	= <i>InR (InR (InR (InR (InR (InR (InL interval))))</i>
∧	<i>classType</i>	= <i>InR (InR (InR (InR (InR (InR (InR class))))</i>

HOL Constant

**rightType** : *Item LIST* → *ColSpec LIST* → *Bool*


---

$\forall is\ css \bullet rightType\ is\ css =$   
 $let\ rightT\ i\ cs =$   
 $let\ right = \lambda t\ v \bullet (t = monoleanType \wedge v = VoidVal)$   
 $\vee (t = booleanType \wedge isBoolVal\ v)$   
 $\vee (t = charsType \wedge isStringVal\ v)$   
 $\vee (t = integerType \wedge isIntVal\ v)$   
 $\vee (t = floatingType \wedge isFloatVal\ v)$   
 $\vee (t = timeType \wedge isTimeVal\ v)$   
 $\vee (t = intervalType \wedge isIntervalVal\ v)$   
 $\vee (t = classType \wedge isClassVal\ v)$   
 $in$   
 $if\ isNullItem\ i$   
 $then\ true$   
 $else\ let\ vi = destValuedItem\ i$   
 $in$   
 $if\ VI\_worth\ vi = sterling$   
 $then\ right\ (CS\_sterlingType\ cs)(VI\_val\ vi)$   
 $else\ right\ (CS\_dinaryType\ cs)(VI\_val\ vi)$   
 $in$   
 $(if\ \#is = 0 \vee \#css = 0$   
 $then\ true$   
 $else\ rightT\ (Hd\ is)\ (Hd\ css) \wedge rightType\ (Tl\ is)\ (Tl\ css))$

HOL Constant

**checkType** : *TableSpec* → *Num* → *Bool* + *Errors*


---

$\forall ts\ n \bullet checkType\ ts\ n =$   
 $let\ itemList = (TS\_rows \ ;\ Map(dataList \ ;\ (elem\ n) \ ;\ Dat\_item))\ ts$   
 $and\ cs = colposns\ ts\ n$   
 $in\ (seqErr\ n\ cs) \ *_5\ (rightType\ itemList)$

We define *rightNull* as an auxiliary function to *CheckNulls*.

HOL Constant

**rightNull** : *Item LIST* → *ColSpec LIST* → *Bool*


---


$$\begin{aligned} \forall is\ css \bullet rightNull\ is\ css = \\ \quad let\ rightN\ i\ cs = ((isNullItem\ i) \wedge CS\_nullType\ cs) \\ \quad in \\ \quad (if\ \#is = 0 \vee \#css = 0 \\ \quad then\ true \\ \quad else\ rightN\ (Hd\ is)\ (Hd\ css) \wedge rightNull\ (Tl\ is)\ (Tl\ css)) \end{aligned}$$

HOL Constant

**checkNulls** : *TableSpec* → *Num* → *Bool* + *Errors*


---


$$\begin{aligned} \forall ts\ n \bullet checkNulls\ ts\ n = \\ \quad let\ itemList = (TS\_rows\ \% Map(dataList\ \% (elem\ n)\ \% Dat\_item))\ ts \\ \quad and\ cs = colposns\ ts\ n \\ \quad in\ (seqErr\ n\ cs) *5\ (rightNull\ itemList) \end{aligned}$$

## 10 SYNTACTIC FUNCTIONS - CONTINUED

We continue with the translation of the syntactic functions from section 8.

HOL Constant

**Col\_name** : *Col\_name* → *Ide*


---


$$\forall i \bullet Col\_name\ (denote\_col\_name\ i) = i$$

HOL Constant

**From\_spec** : *From\_spec* → *State* → *Tuple LIST* + *Errors*


---


$$\begin{aligned} \forall t\ cn\ st \bullet \\ \quad From\_spec\ (from\ t)\ st = projectTuples\ st\ t\ [] \\ \wedge \quad From\_spec\ (correlate\_from\ cn\ t)\ st = projectTuples\ st\ t\ (seq\ 1\ cn) \end{aligned}$$

HOL Constant

**From\_name** : *From\_spec* → *Tab*


---


$$\begin{aligned} \forall t\ cn \bullet \\ \quad From\_name\ (from\ t) = Table\_spec\ t \\ \wedge \quad From\_name\ (correlate\_from\ cn\ t) = Table\_spec\ t \end{aligned}$$

We supply the signature for a function from *Num* to *Int*.

HOL Constant

<b>Num_to_Int</b> : <i>Num</i> → <i>Int</i>
<i>true</i>

We give a function *check\_where\_complete1* which can be applied in the case where the first argument is not of type *Bool*, but of type *Bool* + *Errors*.

HOL Constant

<b>check_where_complete1</b> : <i>Bool</i> + <i>Errors</i> → <i>Bool</i> → <i>Data</i> → <i>Data</i> + <i>Errors</i>
$\forall be\ b\ d \bullet \text{check\_where\_complete1}\ be\ b\ d =$ $\text{if isVal be then check\_where\_complete (destVal be) b d}$ $\text{else giveError(destError be)}$

We give *all\_false* used in the definition of *Tuple\_list\_complete* as an auxiliary function.

HOL Constant

<b>all_false</b> : <i>Bool LIST</i> → <i>Bool</i>
$\forall bs \bullet \text{all\_false}\ bs = \text{if } \#bs = 0 \text{ then true}$ $\text{else if Hd bs = true then false}$ $\text{else all\_false (Tl bs)}$

Similarly *same* which is used as an auxiliary function to *eliminate*, defined locally in *Tuple\_list<sub>p</sub>* and *processUpdate* .

HOL Constant

<b>same</b> : <i>Bool LIST</i> → <i>Bool</i> + <i>Errors</i>
$\forall bs \bullet \text{same}\ bs =$ $\text{if } \#bs = 0 \text{ then giveVal true}$ $\text{else if } \#bs = 1 \text{ then giveVal(Hd bs)}$ $\text{else if (Hd bs) = destVal(same (Tl bs)) then giveVal(Hd bs)}$ $\text{else giveError(seq 1 ambiguousHaving)}$

The functions *Value*, *Tuple\_list\_complete*, *Tuple\_list* and *Select\_list* are mutually recursive. In order to simplify the specifications, we first define paramaterised versions of these functions.

We define values *monop*, *binop*, *set\_func\_all*, *all\_binop* and *classification*, and tuple lists *all\_tuples* and *evaluate* separately to simplify the specifications of *Value<sub>p</sub>* and *Tuple\_list<sub>p</sub>*.

HOL Constant

$$\mathbf{Value\_monop} : (Value \rightarrow State \rightarrow Env \rightarrow Data + Errors) \rightarrow$$

$$Value \rightarrow State \rightarrow Env \rightarrow Data + Errors$$

$$\forall value\ st\ e\ op\ v \bullet$$

$$Value\_monop\ value\ (monop\ op\ v)\ st\ e =$$

$$(let\ v' = value\ v\ st\ e$$

$$in$$

$$if\ isVal\ v'\ then\ giveVal(apply\ op\ (seq\ 1\ (destVal\ v')))$$

$$else\ giveError(destError\ v'))$$

HOL Constant

$$\mathbf{Value\_binop} : (Value \rightarrow State \rightarrow Env \rightarrow Data + Errors) \rightarrow$$

$$Value \rightarrow State \rightarrow Env \rightarrow Data + Errors$$

$$\forall value\ st\ e\ op\ v_1\ v_2 \bullet$$

$$Value\_binop\ value\ (binop\ op\ (v_1, v_2))\ st\ e =$$

$$(let\ v'_1 = value\ v_1\ st\ e\ and\ v'_2 = value\ v_2\ st\ e$$

$$in$$

$$if\ isVal\ v'_1\ then$$

$$if\ isVal\ v'_2\ then$$

$$giveVal(apply\ op\ (Cons(destVal\ v'_1)[destVal\ v'_2]))$$

$$else\ giveError(destError\ v'_2)$$

$$else\ if\ isVal\ v'_2\ then$$

$$giveError(destError\ v'_1)$$

$$else\ giveError((destError\ v'_1) \wedge (destError\ v'_2)))$$

HOL Constant

$$\mathbf{Value\_set\_func\_all} : (Value \rightarrow State \rightarrow Env \rightarrow Data + Errors) \rightarrow$$

$$Value \rightarrow State \rightarrow Env \rightarrow Data + Errors$$

$$\forall value\ st\ e\ op\ v \bullet$$

$$Value\_set\_func\_all\ value\ (set\_func\_all\ op\ v)\ st\ e =$$

$$(let\ all\_res =$$

$$(promote\ \lambda\ t \bullet value\ v\ st\ (MkEnv\ t\ (E\_group\ e)))(E\_group\ e)$$

$$in$$

$$if\ isVal\ all\_res$$

$$then\ giveVal((apply\ op)(destVal\ all\_res))$$

$$else\ giveError\ (destError\ all\_res))$$

HOL Constant

---

**Value\_all\_binop** : (*Tuple\_list* → *State* → *Env* → *Bool* + *Errors*)  
 → (*Tuple\_list* → *State* → *Env* → *Data LIST LIST* + *Errors*)  
 → (*Value* → *State* → *Env* → *Data* + *Errors*)  
 → *Value* → *State* → *Env* → *Data* + *Errors*

---

∀ *t\_l\_c t\_l value st e op v tl* •

*Value\_all\_binop t\_l\_c t\_l value (all\_binop op v tl) st e* =  
 (*let tlc = t\_l\_c tl st e*  
 and *calc* =  
   *let ans = value v st e*  
   in  
   *promote*(λ*d*• *if isVal ans*  
     *then giveVal(apply op ([destVal ans] ∩ [d]))*  
     *else giveError(destError ans)*)  
 in  
 ((((((*t\_l tl st e*) \*<sub>2</sub> *one\_col*) \* *calc*) \*<sub>5</sub> *apply And*) \*  
   *take\_data*) \* (*check\_where\_complete1 tlc true*))

HOL Constant

---

**Value\_classification** : *Value* → *State* → *Env* → *Data* + *Errors*

---

∀ *st e cs* •

*Value\_classification (classification cs) st e* =  
 (*let ce = (((E\_row e)(Col\_spec cs)) \*<sub>5</sub> Dat\_class)*  
 in  
*if isVal ce*  
*then giveVal(make\_data (ClassVal(destVal ce)))*  
*else giveError(destError ce)*)

HOL Constant

---

**Value<sub>p</sub>** : (*Tuple\_list* → *State* → *Env* → *Bool* + *Errors*)  
 → (*Tuple\_list* → *State* → *Env* → *Data LIST LIST* + *Errors*) →  
*Value* → *State* → *Env* → *Data* + *Errors*

---

$\forall t\_l\_c\ t\_l\ st\ e\ i\ s\ r\ m\ int\ cl\ c\ op\ v\ v_1\ v_2\ tl\ cs \bullet$   
*Value<sub>p</sub>* *t\_l\_c t\_l* *denote\_null* *st e* = *giveVal*(*MkData* *userClearance*(*NullItemItem* *null*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *denote\_void* *st e* = *giveVal*(*make\_data* *VoidVal*)  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *denote\_true* *st e* = *giveVal*(*make\_data* (*BoolVal* *true*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *denote\_false* *st e* = *giveVal*(*make\_data* (*BoolVal* *false*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_integer* *i*) *st e* = *giveVal*(*make\_data* (*IntVal* *i*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_string* *s*) *st e* = *giveVal*(*make\_data* (*StringVal* *s*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_float* *r*) *st e* = *giveVal*(*make\_data* (*FloatVal* *r*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_time* *m*) *st e* = *giveVal*(*make\_data* (*TimeVal* *m*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_interval* *int*) *st e* = *giveVal*(*make\_data* (*IntervalVal* *int*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_class* *cl*) *st e* = *giveVal*(*make\_data* (*ClassVal* *cl*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*denote\_code* *c*) *st e* = *giveVal*(*make\_data* (*CodeVal* *c*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*monop* *op v*) *st e* =  
*Value\_monop*(*Value<sub>p</sub>* *t\_l\_c t\_l*)(*monop* *op v*) *st e*  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*binop* *op (v<sub>1</sub>,v<sub>2</sub>)*) *st e* =  
*Value\_binop* (*Value<sub>p</sub>* *t\_l\_c t\_l*) (*binop* *op (v<sub>1</sub>,v<sub>2</sub>)*) *st e*  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*set\_func\_all* *op v*) *st e* =  
*Value\_set\_func\_all* (*Value<sub>p</sub>* *t\_l\_c t\_l*)(*set\_func\_all* *op v*) *st e*  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *count\_all* *st e* =  
*giveVal*(*make\_data*(*IntVal*(*Num\_to\_Int*( $\#(E\_group\ e)$ ))))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*all\_binop* *op v tl*) *st e* =  
*Value\_all\_binop* *t\_l\_c t\_l* (*Value<sub>p</sub>* *t\_l\_c t\_l*) (*all\_binop* *op v tl*) *st e*  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*contents* *cs*) *st e* = (*E\_row* *e*)(*Col\_spec* *cs*)  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* (*classification* *cs*) *st e* = *Value\_classification*(*classification* *cs*) *st e*  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *user* *st e* = *giveVal*(*make\_data* (*StringVal* *userName*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *clearance* *st e* = *giveVal*(*make\_data* (*ClassVal* *userClearance*))  
 $\wedge$  *Value<sub>p</sub>* *t\_l\_c t\_l* *current\_time* *st e* = *giveVal*(*make\_data* (*TimeVal* *timeNow*))

---

HOL Constant

---

**Tuple\_list\_complete<sub>p</sub>** : (*Value* → *State* → *Env* → *Data* + *Errors*)  
→ *Tuple\_list* → *State* → *Env* → *Bool* + *Errors*

---

∀ *value st e sel from\_list where group having* •

*Tuple\_list\_complete<sub>p</sub> value (all\_tuples sel from\_list where group having) st e =*  
*(let froms = (promote (λfr•From\_spec fr st) \* join)*  
*and ws g = promote(λt•value where st (MkEnv t g) \*<sub>5</sub> isNotCleared)g*  
*in*  
*(froms from\_list \* ws) \*<sub>5</sub> all\_false)*

∧ *Tuple\_list\_complete<sub>p</sub> value (distinct\_tuples sel from\_list where group having) =*  
*Tuple\_list\_complete<sub>p</sub> value (all\_tuples sel from\_list where group having)*

∧ *Tuple\_list\_complete<sub>p</sub> value (evaluate sel from\_list where group having) =*  
*Tuple\_list\_complete<sub>p</sub> value (all\_tuples sel from\_list where group having)*

HOL Constant

---

**Tuple\_list\_all\_tuples** : (*Value* → *State* → *Env* → *Data* + *Errors*)  
 → (*Select\_list* → *State* → *Env* → *Tab LIST* → *Data LIST* + *Errors*)  
 → *Tuple\_list* → *State* → *Env* → *Data LIST LIST* + *Errors*

---

∀ *value s\_l st e sel from\_list where group having* •  
*Tuple\_list\_all\_tuples value s\_l(all\_tuples sel from\_list where group having) st e =*  
*(let froms = (promote (λfr•From\_spec fr st) \* join)*  
*and selr env = (let tabs = Map From\_name from\_list*  
*in*  
*(s\_l sel st env tabs))*  
*and outer e = (Map(λei• MkEnv((E\_row e)\*\*(E\_row ei))((E\_row e)\*\*(E\_group ei))))*  
*and where\_filter ts =*  
*(let ws g =*  
*let ev t =*  
*let elim =*  
*let eliminate = (λ s env h•(promote*  
*((((λt•value h s (MkEnv t (E\_group env)))) \* seq 1)*  
*\* applyNot) \*<sub>1</sub> resultBool)(E\_group env)) \* same)*  
*in*  
*eliminate st (MkEnv t g) having*  
*in*  
*if isError elim then elim*  
*else if (destVal elim) = true then (giveVal false)*  
*else (value where st (MkEnv t g)) \* resultBool*  
*in*  
*promote ev g*  
*in*  
*(ts \*<sub>7</sub> ((ws ts) \*<sub>5</sub> extract)))*  
*in*  
*((((froms from\_list) \* where\_filter) \*<sub>5</sub> (engroup group)) \*<sub>5</sub> (outer e)) \* promote selr)*

HOL Constant

**Tuple\_list\_evaluate** : (*Value* → *State* → *Env* → *Data* + *Errors*)  
 → (*Select\_list* → *State* → *Env* → *Tab LIST* → *Data LIST* + *Errors*)  
 → *Tuple\_list* → *State* → *Env* → *Data LIST LIST* + *Errors*

∀ *value s\_l st e sel from\_list where group having* •  
*Tuple\_list\_evaluate value s\_l (evaluate sel from\_list where group having) st e =*  
 (*let froms = (promote (λfr• From\_spec fr st) \* join)*  
*and selg env =*  
   (*let tabs = Map From\_name from\_list*  
   *in*  
   (*if isVal(s\_l sel st env tabs)*  
   *then giveVal(MkGroupedResult(destVal*  
     (*s\_l sel st env tabs))(E\_group env)*  
   *else giveError(destError(s\_l sel st env tabs))*))  
*and outer e = ( Map(λei• MkEnv ((E\_row e)\*\*(E\_row ei))*  
   (*(E\_row e)\*\*(E\_group ei))*))  
*and ev grs = if distinct(Map G\_group grs) = Map G\_group (distinct grs)*  
   *then giveVal(Map G\_res grs)*  
   *else giveError(seq 1 ambiguousEvaluate)*  
*and where\_filter ts =*  
   (*let ws g = promote(λt•value where st (MkEnv t g) \* resultBool)g*  
   *in*  
   (*ts \*7 ((ws ts) \*5 extract)*))  
*in*  
 ((((((*froms from\_list*) \* *where\_filter*) \*5 (*engroup group*)) \*5  
 (*outer e*)) \* *promote selg*) \* *ev*)

HOL Constant

**Tuple\_list\_p** : (*Value* → *State* → *Env* → *Data* + *Errors*)  
 → (*Select\_list* → *State* → *Env* → *Tab LIST* → *Data LIST* + *Errors*)  
 → *Tuple\_list* → *State* → *Env* → *Data LIST LIST* + *Errors*

∀ *value s\_l st e sel from\_list where group having* •  
*Tuple\_list\_p value s\_l (all\_tuples sel from\_list where group having) st e =*  
   *Tuple\_list\_all\_tuples value s\_l (all\_tuples sel from\_list where group having) st e*  
 ∧ *Tuple\_list\_p value s\_l (distinct\_tuples sel from\_list where group having) st e =*  
   (*(Tuple\_list\_p value s\_l (all\_tuples sel from\_list where group having)*  
   *st e) \*5 distinct*)  
 ∧ *Tuple\_list\_p value s\_l (evaluate sel from\_list where group having) st e =*  
   *Tuple\_list\_evaluate value s\_l (evaluate sel from\_list where group having) st e*

HOL Constant

$$\mathbf{Select\_list}_p : (Value \rightarrow State \rightarrow Env \rightarrow Data + Errors) \\ \rightarrow Select\_list \rightarrow State \rightarrow Env \rightarrow Tab\ LIST \rightarrow Data\ LIST + Errors$$
 $\forall$  value st e vl ts •
$$Select\_list_p\ value\ all\_columns\ st\ e = \\ (let\ vals = denote\_col\_spec\ ;\ contents\ ;\ (\lambda v \bullet value\ v\ st\ e) \\ and\ col\ t = enseq(((lookup\ st\ t\ * colposns) * CS\_ide) * (\lambda i \bullet t \wedge [i])) \\ in \\ ((Map\ col) ; Flat) ; (promote\ vals))$$

$$\wedge Select\_list_p\ value\ (select\_values\ vl)\ st\ e\ ts = (promote(\lambda v \bullet value\ v\ st\ e)\ vl)$$
Now we define the syntactic functions *Value*, *Tuple\_list\_complete*, *Tuple\_list* and *Select\_list*.

HOL Constant

$$\mathbf{Value} : Value \rightarrow State \rightarrow Env \rightarrow Data + Errors; \\ \mathbf{Tuple\_list\_complete} : Tuple\_list \rightarrow State \rightarrow Env \rightarrow Bool + Errors; \\ \mathbf{Tuple\_list} : Tuple\_list \rightarrow State \rightarrow Env \rightarrow Data\ LIST\ LIST + Errors; \\ \mathbf{Select\_list} : Select\_list \rightarrow State \rightarrow Env \rightarrow Tab\ LIST \rightarrow Data\ LIST + Errors$$

$$Value = Value_p\ Tuple\_list\_complete\ Tuple\_list$$

$$\wedge Tuple\_list\_complete = Tuple\_list\_complete_p\ Value$$

$$\wedge Tuple\_list = Tuple\_list_p\ Value\ Select\_list$$

$$\wedge Select\_list = Select\_list_p\ Value$$

HOL Constant

---

**Set\_clause** : *Set\_clause* → *State* → *Env* → *Ide* → *Update* + *Errors*


---

 $\forall st\ e\ col\ vv\ vc\ id \bullet$ 

*Set\_clause* (*set\_value col vv*) *st e id* =  
 (let *column* = *Col\_name col*  
 and *exp* = *Value vv st e*  
 in  
 if *id* = *column*  
 then (*exp* \* *take\_data*) \*<sub>5</sub> *DataUpdate*  
 else *giveError(seq 1 error)*)

$\wedge$  *Set\_clause* (*set\_class col vc*) *st e id* =  
 (let *column* = *Col\_name col*  
 and *exp* = *Value vc st e*  
 in  
 if *id* = *column*  
 then (*exp* \* *resultClass*) \*<sub>5</sub> *ClassUpdate*  
 else *giveError(seq 1 error)*)

$\wedge$  *Set\_clause* (*set\_class\_and\_value col vv vc*) *st e id* =  
 (let *column* = *Col\_name col*  
 and *value* = (((*Value vv st e*) \* *take\_data*) \*<sub>5</sub> *Dat\_item*)  
 and *clas* = ((*Value vc st e*) \* *resultClass*)  
 in  
 if *id* = *column*  
 then ((*clas,value*) \* *MkData*) \*<sub>5</sub> *DataUpdate*  
 else *giveError(seq 1 error)*)

HOL Constant

---

**Classified\_value** : *Classified\_value* → *State* → *Env* → *Data* + *Errors*


---

 $\forall st\ e\ v\ vc \bullet$ 

$$\begin{aligned} & \text{Classified\_value}(\text{classify } v\ vc)\ st\ e = \\ & \quad (\text{let } resv = \text{Value } v\ st\ e \\ & \quad \text{and } resc = \text{Value } vc\ st\ e \\ & \quad \text{in} \\ & \quad ((resc *_2 \text{resultClass}), ((resv * \text{take\_data}) *_5 \text{Dat\_item})) * \text{MkData}) \end{aligned}$$
 $\wedge$ 

$$\begin{aligned} & \text{Classified\_value}(\text{classify\_default } v)\ st\ e = \\ & \quad (\text{let } \text{combine } c\ d = \text{MkData } c\ (\text{Dat\_item } d) \\ & \quad \text{and } \text{newDat} = ((\text{Value } v\ st\ e) *_2 \\ & \quad \quad (\lambda d \bullet \text{if } \text{isCleared } d \text{ then } \text{giveVal } d \\ & \quad \quad \quad \text{else } \text{giveError}(\text{seq } 1\ \text{notCleared}))) \\ & \quad \text{and } \text{newClass} = \text{userClearance} \\ & \quad \text{in} \\ & \quad (\text{newDat} *_5 (\text{combine } \text{newClass})) \end{aligned}$$

HOL Constant

---

**Insert\_list** : *Insert\_list* → *State* → *Data LIST LIST* + *Errors*


---

 $\forall st\ tl\ cvl\ il_1\ il_2 \bullet$ 

$$\begin{aligned} & \text{Insert\_list}(\text{insert\_tuples } tl)\ st = \\ & \quad (\text{let } (\text{check}:\text{Data } \text{LIST } \text{LIST} \rightarrow \text{Data } \text{LIST } \text{LIST} + \text{Errors})\ d = \\ & \quad \quad (\text{let } \text{tlc} = \text{Tuple\_list\_complete } tl\ st\ \text{emptyEnv} \\ & \quad \quad \text{in} \\ & \quad \quad \text{if } \text{isVal } \text{tlc} \text{ then} \\ & \quad \quad \quad \text{if } \text{destVal } \text{tlc} = \text{true} \text{ then } \text{giveVal } d \\ & \quad \quad \quad \text{else } \text{giveError}(\text{seq } 1\ \text{notCleared}) \\ & \quad \quad \quad \text{else } \text{giveError}(\text{seq } 1\ \text{notCleared}) \\ & \quad \quad \text{in } (\text{Tuple\_list } tl\ st\ \text{emptyEnv}) * \text{promote}(\text{promote } \text{take\_data})) \end{aligned}$$
 $\wedge$ 

$$\begin{aligned} & \text{Insert\_list}(\text{tuple } cvl)\ st = \\ & \quad (\text{let } d\_or\_err = \text{promote } (\lambda cv \bullet \text{Classified\_value } cv\ st\ \text{emptyEnv})\ cvl \\ & \quad \text{in} \quad \text{if } \text{isVal } d\_or\_err \text{ then } \text{giveVal}(\text{seq } 1\ (\text{destVal } d\_or\_err)) \\ & \quad \quad \text{else } \text{giveError}(\text{destError } d\_or\_err)) \end{aligned}$$
 $\wedge$ 

$$\begin{aligned} & \text{Insert\_list}(\text{union } il_1\ il_2)\ st = \\ & \quad (\text{Insert\_list } il_1\ st \ \&_4 \ \text{Insert\_list } il_2\ st) \end{aligned}$$

We separate the *processQuery* function into four parts, corresponding to the type of query.

For an insert query, a list of relations from *Num* to *Data*, one relation for each new row, will be passed to the update function, hence there is no need for reordering. Classification of the new rows is also achieved by the update function. Only those columns that the user is cleared to see are visible.

If the integrity checks carried out in *processInsert* and *processUpdate* on the hidden state of the database give rise to errors then the update will fail because the function *updateState*, defined in [3], only allows an insert, delete or update query to update the database if no errors have been reported. The error *wrongWidth* is not in the domain of possible errors - replaced by *tooWide*.

We define *processIntegrity* as an auxiliary function.

HOL Constant

**processIntegrity** : *TableSpec* → *Errors*

---

$\forall t \bullet$  *processIntegrity* *t* =  
 (if (checkGroup *t* CC\_uniform checkUniform) = true  
 then noErrors else seq 1 nonUniformValues)  
 $\wedge$  (if (checkGroup *t* CC\_unique checkUniqueness) = true  
 then noErrors else seq 1 nonUniqueValues)  
 $\wedge$  (if (checkIntegrity *t* checkFieldClasses) = true  
 then noErrors else seq 1 fieldClassOutOfRange)  
 $\wedge$  (if (checkIntegrity *t* checkType) = true  
 then noErrors else seq 1 wrongType)  
 $\wedge$  (if (checkIntegrity *t* checkNulls) = true  
 then noErrors else seq 1 noNulls)

HOL Constant

---

```

processInsert : Class → Table_spec → Col_name LIST → Insert_list → State →
                Effect + Errors

```

---

```

∀ c t cns il st • processInsert c t cns il st =
  let spec = lookup st (Table_spec t)
  and givenNames = Map Col_name cns
in  let defaults = ((spec *3 colposns) * CS_default)
in  let givenNums = promote ((spec *3 colspecs) * CS_posn)givenNames
in  let colNums = Map CS_posn (enseq(spec * colposns))
in  let map nums ds n =      if isVal(find nums n)
                             then giveVal(n,elem (destVal(find nums n))ds)
                             else if isVal (defaults n)
                             then giveVal(n,destVal(defaults n))
                             else giveError(destError(defaults n))
in  let addDefaults ds = if #givenNames = 0
                             then promote (map colNums ds)colNums
                             else if isVal givenNums
                             then promote (map (destVal givenNums) ds)colNums
                             else giveError(destError givenNums)
in  let width ds =      if #givenNames = 0
                             then if #colNums = #ds
                                 then giveVal ds
                                 else giveError(seq 1 tooWide)
                             else if #givenNames = #ds
                                 then giveVal ds
                                 else giveError(seq 1 tooWide)
in  let effect =      if isVal spec
                             then if TS_maxRow (destVal spec) dominates userClearance
                                 then giveError(seq 1 rowClassTooLow)
                             else (Insert_list il st) *2 promote (width *1 (addDefaults * Elems))
                             else giveError(destError spec)
in  if isError effect
    then giveError (destError effect)
    else
      let newSpec = replaceRows (destVal spec)((TS_rows (destVal spec)) ^
                                         (Map (MkRow c) (destVal effect)))
in  let integrity = processIntegrity newSpec
in  if ¬(#integrity = 0)
    then giveError integrity
    else giveVal(InsertEffect(Table_spec t,destVal effect))

```

---

A set of rows to be deleted will only be passed on to *updateState* if there are no errors.

HOL Constant

```

processDelete : From_spec → Value → State → Effect + Errors
-----
∀ f w st • processDelete f w st =
  let table = From_name f
in   let spec = lookup st table
in   let ts = From_spec f st
in   let where = if isVal ts
          then promote(λt•Value w st (MkEnv t (destVal ts)))(destVal ts)
          else giveError(destError ts)
in   let whereBools = (where * (promote resultBool))
in   let complete = if isError where then true
          else checkComplete (destVal where)
in   let doomed = if isError whereBools
          then giveError(destError whereBools)
          else giveVal(Dom(ListRel(destVal whereBools) ▷ {true}))
in   if isError doomed
      then giveError(destError doomed)
      else if complete then giveVal(DeleteEffect(From_name f, destVal doomed))
      else giveError(seq 1 mayNotBeComplete)

```

We provide a function that converts a total function of type column number to update to the appropriate partial function.

HOL Constant

```

convert : (Num → Update + Errors) → (Num ↔ Update) + Errors
-----
∀ f • convert f =
  if ∃ n • isError(f n) ∧ (¬ (destError(f n) = seq 1 error))
  then giveError(RelList(Enumerate(∪{x|∃y•
    isError(f y) ∧ (¬ (destError(f y) = seq 1 error)) ∧ x = Elems(destError(f y))})))
  else giveVal{(id,upd)|f id = giveVal upd}

```

We provide a function which takes a list of rows and updates and returns the updated list of rows.

HOL Constant

---

**updateRowList** : *Row LIST*  $\rightarrow$  (*Num*  $\leftrightarrow$  (*Num*  $\leftrightarrow$  *Update*))  $\rightarrow$  *Row LIST*


---

$\forall$  *rl nnu* • *updateRowList* *rl nnu* =

*let* *updateD* ((*u:Update*),(*d:Data*)) =

*if* *isItem* *u* *then* *MkData* (*Dat\_class* *d*)(*destItem* *u*)

*else if* *isClass* *u* *then* *MkData* (*destClass* *u*)(*Dat\_item* *d*)

*else* *destData* *u*

*in* *let* *updateR* ((*nu:Num*  $\leftrightarrow$  *Update*),(*r:Row*)) = *MkRow*

    (*R\_exist* *r*)(*R\_data* *r*)  $\oplus$  (*RelCombine* *nu* (*R\_data* *r*))  $\%$  *Graph* *updateD*)

*in* *RelList*((*ListRel* *rl*)  $\oplus$  (*RelCombine* *nnu* (*ListRel* *rl*)  $\%$  *Graph* *updateR*))

HOL Constant

---

**processUpdate** : *From\_spec*  $\rightarrow$  *Set\_clause LIST*  $\rightarrow$  *Value*  $\rightarrow$  *Col\_spec LIST*  
 $\rightarrow$  *Value*  $\rightarrow$  *State*  $\rightarrow$  *Effect* + *Errors*


---

$\forall$  *f ss w g h st* • *processUpdate* *f ss w g h st* =

*let* *table* = *From\_name* *f*

*in* *let* *spec* = *lookup* *st* *table*

*in* *let* *ts* = *From\_spec* *f* *st*

*in* *let* *whereResults* = *let* *ev* *e* =

*let* *elim* =

*let* *eliminate* =

        ( $\lambda$  *s env h* • (*promote*

            ((( $\lambda$  *t* • *Value* *h* *s* (*MkEnv* *t* (*E\_group* *env*)))) \* *seq* 1)

            \* *applyNot*) \*<sub>1</sub> *resultBool*)(*E\_group* *env*)) \* *same*)

*in*

*eliminate* *st* *e* *h*

*in*

    (*if* *isError* *elim* *then* (*giveError*(*destError* *elim*))

*else if* (*destVal* *elim*) = *true*

*then* (*giveVal* (*make\_data*(*BoolVal* *false*)))

*else* (*Value* *w* *st* *e*))

*in*

    (*if* *isVal* *ts* *then* *promote* *ev* (*engroup* *g* (*destVal* *ts*))

*else* *giveError*(*destError* *ts*))

*in* *let* *whereBools* = (*whereResults* \* (*promote* *resultBool*))

*in* *let* *complete* = *if* *isError* *whereResults* *then* *true*

*else* *checkComplete* (*destVal* *whereResults*)

*in* *let* *sets* *e* = (( $\lambda$  *c* • *giveError*(*seq* 1 *error*)) & *Map*( $\lambda$  *s* • *Set\_clause* *s* *st* *e*)) *ss*)

*in* *let* *getEnv* *n* = *if* *isVal* *ts* *then* *giveVal*(*MkEnv*(*Nth* (*destVal* *ts*)*n*)(*destVal* *ts*))

*else* *giveError*(*destError* *ts*)

---

```

in   let rowUpd n = convert(((spec *3 colposns) * CS_ide) *1 (getEnv n * sets))
in   let where =   if isError whereBools
                  then giveError(destError whereBools)
                  else giveVal(Dom(ListRel(destVal whereBools) ▷ {true}))
in   if isError where
    then giveError(destError where)
    else if complete
        then let us = {(n,nu)|n ∈ destVal where ∧ nu = rowUpd n}
              in   if ∃ pr • pr ∈ us ∧ isError(Snd pr)
                    then giveError(RelList(Enumerate(∪{x|∃y•
                                                    y ∈ us ∧ x = Elems(destError(Snd y))})))
                    else
                        let ups = {(n,nu)|n ∈ destVal where ∧ nu = destVal(rowUpd n)}
                        in   let oldRows = TS_rows (destVal spec)
                              in   let newSpec = replaceRows (destVal spec)
                                    (updateRowList oldRows ups)
                        in   let integrity = processIntegrity newSpec
                              in   if ¬(#integrity = 0)
                                    then giveError integrity
                                    else giveVal(UpdateEffect(table,ups))
        else giveError(seq 1 mayNotBeComplete)

```

HOL Constant

---

**processSelect** : *Tuple\_list* → *State* → *Effect* × *Errors*

---

```

∀ tl st • processSelect tl st =
  let retrieved = Tuple_list tl st emptyEnv
  and tlc = Tuple_list_complete tl st emptyEnv
in   if isError retrieved
    then (SelectEffect[],destError retrieved)
    else if isVal tlc
        then if(destVal tlc) = true
              then (SelectEffect(destVal retrieved),[])
              else (SelectEffect(destVal retrieved),seq 1 mayNotBeComplete)
    else (SelectEffect[],destError tlc)

```

We provide a null update effect to be returned by *processQuery* in the case where the result of the query produced errors.

HOL Constant

---

**nullUpdate** : *Effect*

---

*true*

---

HOL Constant

---

**processQuery** :  $Query \times Class \times State \rightarrow Effect \times Errors$ 


---

 $\forall c\ st\ t\ cns\ il\ f\ w\ ss\ g\ h\ tl \bullet$ 

$$\begin{aligned} processQuery((insert\ t\ cns\ il),c,st) = \\ & (let\ iorerr = processInsert\ c\ t\ cns\ il\ st \\ & \quad in \\ & \quad if\ isVal\ iorerr \\ & \quad then\ (destVal\ iorerr,[]) \\ & \quad else\ (nullUpdate,destError\ iorerr)) \end{aligned}$$

$$\wedge\ processQuery((delete\ f\ w),c,st) = \\ & (let\ dorerr = processDelete\ f\ w\ st \\ & \quad in \\ & \quad if\ isVal\ dorerr \\ & \quad then\ (destVal\ dorerr,[]) \\ & \quad else\ (nullUpdate,destError\ dorerr))$$

$$\wedge\ processQuery((update\ f\ ss\ w\ g\ h),c,st) = \\ & (let\ uorerr = processUpdate\ f\ ss\ w\ g\ h\ st \\ & \quad in \\ & \quad if\ isVal\ uorerr \\ & \quad then\ (destVal\ uorerr,[]) \\ & \quad else\ (nullUpdate,destError\ uorerr))$$

$$\wedge\ processQuery((select\ tl),c,st) = processSelect\ tl\ st$$

## 11 CLOSING DOWN

The following ProofPower instruction restores the previous proof context.

SML

```
|pop_pc();
```

## 12 INDEX

**	24	<i>correlate_from</i>	26
* <sub>1</sub>	13	<i>count_all</i>	25
* <sub>2</sub>	13	<i>current_time</i>	25
* <sub>3</sub>	13	<i>dataList</i>	31
* <sub>4</sub>	14	<i>delete</i>	27
* <sub>5</sub>	14	<i>denote_class</i>	25
* <sub>6</sub>	14	<i>denote_code</i>	25
* <sub>7</sub>	14	<i>denote_col_name</i>	26
*	15	<i>denote_col_spec</i>	26
<i>all_binop</i>	25	<i>denote_false</i>	25
<i>all_columns</i>	27	<i>denote_float</i>	25
<i>all_false</i>	36	<i>denote_integer</i>	25
<i>all_tuples</i>	26	<i>denote_interval</i>	25
<i>andBools</i>	30	<i>denote_null</i>	25
<i>andb</i>	30	<i>denote_string</i>	25
<i>And</i>	8	<i>denote_table_spec</i>	26
<i>applyAnd</i>	10	<i>denote_time</i>	25
<i>applyEqual</i>	12	<i>denote_true</i>	25
<i>applyNot</i>	9	<i>denote_void</i>	25
<i>applyOr</i>	11	<i>destBoolVal</i>	5
<i>applyPlus</i>	12	<i>destClassVal</i>	5
<i>apply</i>	13	<i>destCodeVal</i>	5
<i>auxapply</i>	29	<i>destFloatVal</i>	5
<i>binop</i>	25	<i>destIntervalVal</i>	5
<i>booleanType</i>	33	<i>destIntVal</i>	5
<i>charsType</i>	33	<i>destNullItem</i>	4
<i>checkComplete</i>	20	<i>destStringVal</i>	5
<i>checkFieldClasses</i>	32	<i>destTimeVal</i>	5
<i>checkGroup</i>	31	<i>destValuedItem</i>	4
<i>checkIntegrity</i>	32	<i>distinct_tuples</i>	26
<i>checkNulls</i>	35	<i>distinct</i>	18
<i>checkType</i>	34	<i>dominates_w</i>	7
<i>checkUniform</i>	31	<i>elem</i>	32
<i>checkUniqueness</i>	31	<i>emptyEnv</i>	22
<i>check_where_complete1</i>	36	<i>emptyTuple</i>	22
<i>check_where_complete</i>	20	<i>engroup</i>	29
<i>classification</i>	25	<i>enseq</i>	17
<i>Classified_value</i>	45	<i>Env</i>	7
<i>classify_default</i>	26	<i>Equal</i>	8
<i>classify</i>	26	<i>evaluate</i>	26
<i>classType</i>	33	<i>ExceptionData</i>	9
<i>clearance</i>	25	<i>extract</i>	21
<i>colposns</i>	30	<i>E_group</i>	7
<i>colsInGroup</i>	29	<i>E_row</i>	7
<i>colspecs</i>	30	<i>false</i>	4
<i>Col_name</i>	35	<i>fef014</i>	4
<i>Col_spec</i>	27	<i>fillCol</i>	18
<i>Col</i>	6	<i>fillTab</i>	18
<i>cons</i>	30	<i>find</i>	17
<i>contents</i>	25	<i>floatingType</i>	33
<i>convert</i>	48	<i>From_name</i>	35

<i>From_spec</i> .....	35	<i>Plus</i> .....	8
<i>from</i> .....	26	<i>processDelete</i> .....	48
<i>getColPosn</i> .....	19	<i>processInsert</i> .....	47
<i>getData</i> .....	19	<i>processIntegrity</i> .....	46
<i>getDir</i> .....	19	<i>processQuery</i> .....	51
<i>getTab</i> .....	19	<i>processSelect</i> .....	50
<i>gbl</i> .....	7	<i>processUpdate</i> .....	49
<i>GroupedResult</i> .....	6	<i>projectTuples</i> .....	28
<i>G_group</i> .....	6	<i>promote</i> .....	17
<i>G_res</i> .....	6	<i>resultBool</i> .....	21
<i>inRange</i> .....	32	<i>resultClass</i> .....	21
<i>Insert_list</i> .....	45	<i>rightNull</i> .....	35
<i>insert_tuples</i> .....	26	<i>rightType</i> .....	34
<i>insert</i> .....	27	<i>same</i> .....	36
<i>integerType</i> .....	33	<i>Select_list<sub>p</sub></i> .....	43
<i>intervalType</i> .....	33	<i>Select_list</i> .....	43
<i>intPlus</i> .....	11	<i>select_values</i> .....	27
<i>isBoolVal</i> .....	6	<i>select</i> .....	27
<i>isClassVal</i> .....	6	<i>seqErr</i> .....	32
<i>isCleared</i> .....	20	<i>seq</i> .....	17
<i>isCodeVal</i> .....	6	<i>set_class_and_value</i> .....	26
<i>isFloatVal</i> .....	6	<i>set_class</i> .....	26
<i>isIntervalVal</i> .....	6	<i>Set_clause</i> .....	44
<i>isIntVal</i> .....	6	<i>set_func_all</i> .....	25
<i>isNotCleared</i> .....	20	<i>set_value</i> .....	26
<i>isNullItem</i> .....	5	<i>star1</i> .....	22
<i>isStringVal</i> .....	6	<i>star2</i> .....	22
<i>isTimeVal</i> .....	6	<i>star3</i> .....	23
<i>isValuedItem</i> .....	5	<i>starstar1</i> .....	23
<i>jay</i> .....	23	<i>starstar2</i> .....	23
<i>join</i> .....	23	<i>Table_spec</i> .....	27
<i>list_∪</i> .....	15	<i>take_data</i> .....	21
<i>lookup</i> .....	19	<i>test</i> .....	31
<i>lubl</i> .....	7	<i>timeNow</i> .....	7
<i>lub_data</i> .....	8	<i>timeType</i> .....	33
<i>lub_wdata</i> .....	9	<i>true</i> .....	4
<i>lub_wl</i> .....	8	<i>Tuple_list_all_tuples</i> .....	41
<i>lub_w</i> .....	8	<i>Tuple_list_complete<sub>p</sub></i> .....	40
<i>make_data</i> .....	18	<i>Tuple_list_complete</i> .....	43
<i>MaybeResult</i> .....	7	<i>Tuple_list_evaluate</i> .....	42
<i>Maybe</i> .....	6	<i>Tuple_list<sub>p</sub></i> .....	42
<i>maybe</i> .....	6	<i>Tuple_list</i> .....	43
<i>monoleanType</i> .....	33	<i>Tuple</i> .....	6
<i>monop</i> .....	25	<i>tuple</i> .....	26
<i>newData</i> .....	8	<i>union</i> .....	26
<i>noErrors</i> .....	18	<i>updateRowList</i> .....	49
<i>Not</i> .....	8	<i>update</i> .....	27
<i>NullData</i> .....	9	<i>userClearance</i> .....	7
<i>nullUpdate</i> .....	50	<i>userDirectory</i> .....	7
<i>Num_to_Int</i> .....	36	<i>userName</i> .....	7
<i>one_col</i> .....	18	<i>user</i> .....	25
<i>one_result</i> .....	18	<i>Value_all_binop</i> .....	38
<i>Op</i> .....	8	<i>Value_binop</i> .....	37
<i>Or</i> .....	8	<i>Value_classification</i> .....	38

*Value\_monop* . . . . . 37  
*Value\_set\_func\_all* . . . . . 37  
*Value<sub>p</sub>* . . . . . 39  
*Value* . . . . . 43  
*&<sub>1</sub>* . . . . . 15  
*&<sub>2</sub>* . . . . . 16  
*&<sub>3</sub>* . . . . . 16  
*&<sub>4</sub>* . . . . . 16  
*&<sub>5</sub>* . . . . . 28  
*&<sub>6</sub>* . . . . . 29  
*&* . . . . . 17  
*&* . . . . . 29