Project: DRA FRONT END FILTER PROJECT

Title: Table Computations for SWORD

Ref: DS/FMU/FEF/032 *Issue: Revision : 2.1* *Date:* 5 June 2016

Status: Approved *Type:* Specification

Keywords:

Author:

Name	*Location*	*Signature*	*Date*
R. D. Arthan	WIN01		

Authorisation for Issue:

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

Abstract: A specification of the table computations allowed in the Front End implementation of SWORD 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] Michael J.C. Gordon. Mechanising Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Proceedings of the 1988 Banff Conference on Hardware Verification*. Springer-Verlag, 1988.

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

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

[4] DS/FMU/FEF/020. *Specification of Query Transformations in SML (II)*. G.M. Prout, ICL Secure Systems, WIN01.

[5] DS/FMU/FEF/022. *SWORD Front End Architectural Model*. R.D. Arthan, ICL Secure Systems, WIN01.

[6] DS/FMU/FEF/026. *Critical Requirements on the SWORD Query Transformations*. R.D. Arthan, ICL Secure Systems, WIN01.

[7] DS/FMU/FEF/033. *Value Computation Security Proofs*. R.D. Arthan, ICL Secure Systems, WIN01.

[8] DS/FMU/FEF/034. *Phase II Proof Strategy*. R.D. Arthan, ICL Secure Systems, WIN01.

[9] DS/FMU/FEF/036. *Phase II Proof Finale*. R.D. Arthan and G.M. Prout, ICL Secure Systems, WIN01.

[10] *The Specification of Secure SQL*. Simon Wiseman, DRA, 6th July 1992.

[11] *SSQL Transformations*. Simon Wiseman, DRA, 14th January 1993.

0.3 Changes History

Issue 1.1 (23 August 1993) First draft.

Issue 1.23 (9 November 1993) Tidying up.

Issue *Revision : 2.1* **(5 June 2016)** Final approved version.

Issue 2.2 Removed dependency on ICL logo font

2016/04/11 Added some missing indexing brackets.

0.4 Changes Forecast

None.

1 GENERAL

1.1 Scope

This document gives a formal specification of part of the SWORD Front End reducing the security properties given in [6] to lower level properties more closely related to the detailed transformations of [11]. It constitutes part of deliverable D12 of work package 3, as given in the Phase 2 Technical Proposal, [2].

Some of the material in this document was previously included in [6]. Now that the formal development has been further advanced it was felt more appropriate to collect this level of the treatment into a separate document.

1.2 Introduction

[6] gives a formal description of an Execution Model for the Front End Implementation of SWORD. The Execution Model affords a rather more concrete formulation of the critical requirements on the major subsystems of SWORD than the architectural model of [5]. In particular, it separates out the critical requirements (as regards the *select* query) on the SSQL Query Transformation Processor of [5] so that they are completely determined by a model of a "compiler" for TSQL which maps a TSQL query onto the function on tables which it computes.

The purpose of this document is to complete the formal treatment for the Phase 2 work by further reducing the critical requirements on the SSQL Query Transformation Processor by placing a bound on the allowed TSQL queries it produces. This is done using what is in effect a reformulation of the classification computations which underlie the transformations defined in [11] in terms of a relational algebra model of TSQL execution.

2 PRELIMINARIES

The following ProofPower instructions set the context for the proof tools and set up the new theory *fef032*, with parent the theory *fef026* in which the Execution Model is defined.

SML

```
open_theory "fef026";
(force_delete_theory "fef032" handle _ => ());
new_theory"fef032";
push_pc "hol";
```

3 DISCUSSION

3.1 Objectives

The objective of this document is to define a set of operations on the derived tables of [6] in terms of which it is possible to characterise the allowable results of compiling the queries produced by the SSQL Transformation Processor. This is intended to give some insight into the intuitions about operations on tables which motivate the transformations. It is also intended to allow formal reasoning about a reasonably accurate formal model of the transformations.

3.2 Formal Approach

The idea is to identify the primitive operations on tables and on the security information in them which underlie the semantics of SSQL and its implementation via the transformations. This amounts to something like an SSQL analogue of the relational algebra in which security classifications are computed along with the data operations. The critical properties of [6] may then be rephrased to say something like "provided the SSQL Transformation Processor always produces TSQL query sequences which could equally well have been computed by this SSQL relational algebra, then the security policy will be enforced". Since the computation of security classifications can be formalised in a way which reflects the syntactic formulation of the transformations in [11], the proviso here is amenable to informal checking, and as the semantic framework in which these computations are formalised is much simpler than the syntactic one, their information flow properties are amenable to formal reasoning.

The definitions of the computations are formally related to the security policy by using them to characterise a class of table computations for which the risk inputs, as defined in [6], are either empty or have a known form. This is then used to give a sufficient condition for the truth of the assertion $STP \in STP\_secure\_E\ compile$ which plays an important role in [6]. This sufficient condition can be seen to relate more closely, at least informally, to the internal details of the transformations than the definition of STP_secure itself.

3.3 Example

The SSQL analogue of the relational algebra will be formulated using what is called a "semantic embedding", e.g. see[1], of the relevant aspects of the SSQL language. It may be helpful to give a tiny example of this type of approach. For completeness, the example includes two short proofs, but understanding of these should not be necessary for the example to motivate the specifications in the rest of the document.

We consider a language of expressions formed from constants, K, a single variable, v, and addition:

Example Syntax

$$E = K \mid v \mid E, +, E$$

If the variable is to range over the natural numbers, a semantic embedding for this little language might go as follows.

First of all, we identify the domain over which the semantic values of expressions range. In this case, the domain is the set of functions on the natural numbers taking natural number values; after some preliminary red tape, we capture this in a type abbreviation

SML

```
open_theory"fin_set";
new_theory"eg";
```

SML

```
declare_type_abbrev("E", [], ⌜: ℕ → ℕ ⌝);
```

We now define the semantic functions for the three constructors for expressions, arranging to do this so that they compose in a way which corresponds closely to the abstract structure of the syntax:

HOL Constant

$$K : \mathbb{N} \to E;$$
$$v : E;$$
$$P : E \to E \to E$$

$$(\forall n\bullet\ K\ n = \lambda i\bullet\ n)$$
$$\land \quad (v = \lambda i\bullet\ i)$$
$$\land \quad (\forall e_1\ e_2\bullet\ P\ e_1\ e_2 = \lambda i\bullet\ e_1\ i + e_2\ i)$$

So for example, $P\,(K\ 1)\ v$ is the semantic value of the expression $1 + v$.

We may now define the totality of all functions which arise as the semantics of expressions in the little language; we do this by forming the smallest set of functions which contains all the "ground" functions $K\ i$ and v and is closed under the constructor P:

HOL Constant

$$A : E\ \mathbb{P}$$

$$A = \bigcap\{B \mid (\forall n\bullet K\ n \in B) \land v \in B \land (\forall f_1\ f_2\bullet\ f_1 \in B \land f_2 \in B \Rightarrow P\ f_1\ f_2 \in B)\}$$

Now we can derive an induction principle for the set A. Note that the induction principle gives a means of reasoning by "induction over the syntax of the language" without our having to define the syntax!

SML

```
set_pc"hol2";
set_goal([], ⌜
        ∀p:E → BOOL•
                (∀n• p(K n))
        ∧       p v
        ∧       (∀f1 f2• f1 ∈ A ∧ p f1 ∧ f2 ∈ A ∧ p f2 ⇒ p(P f1 f2))
        ⇒       (∀f•f ∈ A ⇒ p f)
⌝);
```

The proof of this is an essentially mechanical reformulation of the definition of A to use properties (boolean functions) rather than sets.

SML

```
a(rewrite_tac[get_spec⌜A⌝] THEN REPEAT strip_tac);
a(POP_ASM_T (strip_asm_tac o rewrite_rule[get_spec⌜A⌝]
                o ∀_elim⌜{g | g ∈ A ∧ p g}⌝)
        THEN all_asm_fc_tac[] THEN all_asm_fc_tac[]);
val A_induction_thm = save_pop_thm"A_induction_thm";
```

The induction principle (expressed as a HOL theorem) may readily be turned into a tactic using a standard ProofPower tactic-generating function:

SML

```
val A_induction_tac = gen_induction_tac1 A_induction_thm;
```

This tactic may now be used to reason about the functions in A. For example, we can prove that all of the functions in the set A can be written in a standard form:

SML

```
set_goal([], ⌜
        ∀f• f ∈ A ⇒ ∃n k•f = λi•n*i + k
⌝);
```

After applying the induction tactic, this goal reduces to three cases as one might expect, and the proofs in each case are straightforward:

SML

```
a(strip_tac THEN A_induction_tac);
(* *** Goal "1" *** *)
                (* K n = λi•0*i + n *)
a(∃_tac⌜0⌝ THEN ∃_tac⌜n⌝ THEN rewrite_tac[get_spec⌜K⌝]);
(* *** Goal "2" *** *)
                (* v = λi•1*i + 0 *)
a(∃_tac⌜1⌝ THEN ∃_tac⌜0⌝ THEN rewrite_tac[get_spec⌜v⌝]);
(* *** Goal "3" *** *)
                (* P(λi• n*i + k)(λi• n'*i + k') = (λi• (n + n')*i + (k + k')) *)
a(∃_tac⌜n + n'⌝ THEN ∃_tac⌜k + k'⌝ THEN asm_rewrite_tac[get_spec⌜P⌝]
        THEN PC_T1 "lin_arith" prove_tac[]);
val A_standard_form_thm = save_pop_thm"A_standard_form_thm";
```

Now we tidy up by removing the theory in which the example was produced.

SML

```
open_theory"fef032";
delete_theory"eg";
```

For what follows, the main feature of the above example is the use of semantic concepts only to define what one might otherwise define by a mixture of syntactic and semantic means ("the set of all computations definable in such-and-such a language"). We will be concerned with information-flow security properties of a simplified model of the SWORD implementation. The semantic functions we will define will bring out the security checks which correspond to each of the constructors of the SSQL language. The intention is that the overall security checks required by the SSQL semantics will then reduce to properties of the individual constructors, in much the same way that the "standard form" property in the example reduces, by an induction principle, to properties of K, v and P.

3.4 Overview

The plan of what follows is similar in broad outline to the example of the previous section. However, because we are dealing with a significant fragment of a real language, the details are quite a lot more complex.

First we must identify the relevant semantic domains. As in [6], we wish to simplify matters by ignoring naming issues. Thus, the rather elaborate means provided by SSQL to name objects are modelled here by numerical indices of columns in tables, or of tables within a list of same. We will also ignore type distinctions in the syntax, although we will try to point out, informally, places where type information may be relevant to security. Of course, the underlying data we work with is the typed data of the SSQL language as formalised in [3], and so, to that extent, the treatment does reflect the type system of SSQL. Furthermore, we are only concerned with a part of the SSQL language. Indeed, some of the syntactic sorts of SSQL are not part of the data manipulation language and are not treated here; moreover, within the data manipulation language, only the *SELECT* query is to be covered by the present work.

After these simplifications, there are really only two semantically distinct syntactic sorts left, namely *values*, denoting the sort of things which appear in the fields of a database row, and, *tables*, denoting whole tables. The correspondence between this simplified view of sorts and that of the SSQL specification of [10] is shown in the following table:

SSQL Sort	Simplified Sort	Remarks
Type	n/a	see above
Clause	n/a	not part of the DML
Constant_value	VALUE	distinction from *Value* not relevant here
Value	VALUE	—
Col_spec	n/a	numeric index used instead
Table_spec	n/a	numeric index used instead
Col_name	n/a	numeric index used instead
From_spec	TABLE	distinction from *Tuple_list* not relevant here
Target_spec	n/a	not part of the *SELECT* query
Set_clause	n/a	not part of the *SELECT* query
Tuple_list	TABLE	distinction from *From_spec* not relevant here
Select_value	VALUE	distinction from *Value* not relevant here
Select_list	TABLE	just a list of *Select_value*s
Query	TABLE	only dealing with *SELECT* here
BoundQuery	n/a	distinction from *Query* not relevant here

The type we use for computations of sort *TABLE* is closely related to the type used to model the TSQL compiler in [6]: in that document the compiler is thought of as a function which given a query produces a function on lists of derived tables; the result returned by the function representing a compiled query is a pair comprising a new derived table and a list of error indicators. Here, on the one hand, for simplicity, we assume that error detection in the TSQL database is disabled (as, indeed, we understand it is likely to be in the SWORD implementation); on the other hand, we need some additional information to keep track of a security-relevant issue which is not currently addressed in [11], namely, a classification to be used when a table computation appears within some other computation, i.e., a nested *SELECT*. This classification represents the clearance required to evaluate the *GROUPBY* and *HAVING* clauses in the nested *SELECT*. We arrive at the type given in the following type abbreviation for the semantics of table computations.

SML

```
declare_type_abbrev("TABLE_COMP", [],
        ⌜:DerTable LIST → (Class × DerTable)⌝);
```

The type used for computations of sort *VALUE*, which we will often refer to as *expressions*, is a little bit more complicated and is forced on us by the SSQL semantics. The simplest expressions just read some fields in a table row and compute from them a class and a data item. The set functions operate on a list of rows determined by the *GROUPBY* clause in the surrounding table computation. The *exists_tuples* and *single_value* expressions operate on a nested *SELECT* and so require access to the list of derived tables which was the operand of the surrounding table computation. We arrive at the type of functions with three arguments given in the following type abbreviation for the semantics of value computations:

SML

```
declare_type_abbrev("VALUE_COMP", [],
        ⌜:(DerTable LIST → DerTableRow LIST→ DerTableRow → (Class × Item))⌝);
```

The semantic functions for the value and table computations are defined in sections 4 and 5 respectively. Section 6 brings these definitions together to define the algebra of relational operations which we will claim has the relevant security properties. Section 7 defines some notions relating the earlier material to the critical properties identified in [6].

3.5 Omissions and Assumptions

The present version of this document omits certain features of the transformations of [11] and relies on various simplifying assumptions. These shortcomings arise from several practical difficulties, and it may be helpful to summarise these here:

1. The model given here effectively suppresses errors arising in the evaluation of queries. It is understood that such errors will actually be suppressed in the SWORD implementation since there are known security risks associated with them. Where it is necessary to model error cases explicitly here, it is done by setting result data to an arbitrary but fixed value. This may be inaccurate in the case of the uniqueness checks made for the *single_value* and *evaluate* forms (and in the auxiliary *CommonValue* used to model part of the semantics of *HAVING* clauses). Elsewhere, this sanitising is only required in cases corresponding to compile/transformation-time errors which happen to be modelled here dynamically, and so is less likely to affect security.
2. The type conversions (*convert* value form) have not been dealt with explicitly in the formal treatment. The only security-relevant aspect of these is the possibility the conversion would lead to a run-time error. Since we are assuming that error-detection is disabled, this value form may be considered to be covered under the treatment of monadic operators in section 4.2.
3. The conversions which are parameterised by a table (*convert_domain* value form) are potential covert channels. Unfortunately, the checks which must be imposed to prevent illicit information flow via this means are not defined in [11]. If the conversion domain tables are taken as part of the fixed structure of the database, then this value form may be considered to be covered under the treatment of monadic operators in section 4.2.
4. The representation of SSQL data items is borrowed from the Phase 1 work on the semantics in [3]; this uses a different treatment of sterling and dinary data from [11] and the more recent issues of [10]. For present purposes, where the later syntax has separate options for selecting or defining data as sterling or dinary, we have just modelled the default (sterling) case. It seems unlikely that amending the present specification to handle the more recent syntax would significantly affect the relevant security issues.
5. The *all_bin_op*, *some_bin_op* value forms and their *_list* variants are not catered for. This is because there were some small problems with the treatment of these which were remarked upon when [4] was being written, and it was agreed at that time to leave these out. Remedying these deficiencies would probably be straightforward and introduce no significant new features as regards security.
6. The *row_existence* value form is omitted from the present draft. It would appear that this form can always be eliminated in favour of an equivalent use of a *joined_row_existence* inside a nested *SELECT* in the *From_spec* of a query (this is certainly true of the model of the language given here, even if syntactic restrictions prohibit it for SSQL proper). Inclusion of *row_existence* would require a slight complication of the semantic domains to allow the extra information it requires about the construction of the current row to be available.
7. The *context* and *parameter* value forms are omitted, since it is assumed that the relevant information has already been bound into the query being processed.
8. The *classify* and *classify_default* value forms, are treated in [11] so as to allow a client to regrade a value arbitrarily. This is not secure. It is understood that the use of these forms is

intended to be restricted to the generation of values for use in update and insert operations, in circumstances where it has already been checked whether the client is cleared to know the value in question. These forms are therefore not properly part of the *SELECT* query and have been omitted.

9. The transformations of [11] are understood to be too lax with nested *SELECT*s. The approach taken here is to associate an overall class with the derived table produced by a nested *SELECT* and to let this propagate up into the surrounding expression so that it can be checked against the client clearance when the computation is complete. It is believed that this approach cannot be implemented with available TSQL implementations — it would most naturally correspond to the transformed query generating some form of exception if the client was not cleared to know the result of the nested table computation, but this cannot be achieved.

   The approach taken here can be viewed as compatible with a treatment of nested *SELECT*s in which the transformations attempt to evaluate the check statically and refuse to allow the query if this static check fails or if it cannot be evaluated at transformation-time. More explicitly, such a treatment would handle nested *SELECT*s in much the same way as top-level ones, but would reject nested *SELECT*s which were such that the optional check query would be used to enforce security at the top level.

10. The semantics of the *SELECT DISTINCT* query require duplicate rows to be eliminated from the result table; the semantics of the *EVALUATE* query require an error to be signalled if the expression being evaluated is not single-valued on each group determined by the *GROUPBY* clause. The transformations of [11] do not, however, introduce any checks that the client is cleared to have the necessary calculations performed. This point has not yet been addressed here, and, while semantic functions for these queries are given following [11], they are commented out of the construction in section 6, since they are known not to be secure.

11. The treatment here of the optional check query in 7 is still rather implicit (as opposed to the data query for which a specific bound is given in terms of table computations). A more explicit treatment of this aspect should be straightforward, but has been deferred until work on proof has begun to validate the current treatment of the critical properties.

4 VALUE COMPUTATIONS

The particular expression forms which model those of TSQL are given in sections 4.1 to 4.16 below. The classifications follow those assigned in [11]. The classification information is taken from $internal\_value_{class}$ in [11] unless otherwise noted below. The order of the treatment is also taken from $internal\_value_{class}$.

4.1 Constant Expression

The function giving constant expressions is parameterised by the classification and value of the constant.

HOL Constant

DenoteConstant : $(Class \times Item) \to VALUE\_COMP$

$\forall ci \bullet \quad DenoteConstant\ ci = \lambda tl\ rl\ r \bullet\ ci$

4.2 Monadic

The monadic forms are parameterised here by the actual item computation to perform and the expression which computes the operand.

HOL Constant

MonOp : $(Item \to Item) \to VALUE\_COMP \to VALUE\_COMP$

$\forall f\ e \bullet \quad MonOp\ f\ e =$
$\quad \lambda tl\ rl\ r \bullet$
$\quad let \quad (c,\ v) = e\ tl\ rl\ r$
$\quad in \quad (c,\ f\ v)$

4.3 Binary

The logical binary operators are treated specially. In particular, iterated *and*s and *or*s are effectively treated by $simplify_{ands}$ and $simplify_{ors}$in [11] as composite operators on lists of expressions, for the purpose of computing the classification. We model this directly here (although this makes misnomers of the names). Thus *BinOpAnd* and *BinOpOr* below are parameterised by lists of expressions to compute the operands to be combined.

We use the following to coerce *Item*s into truth values and vice versa:

HOL Constant

ItemBool : $Item \to Bool$

$\forall v \bullet \quad ItemBool\ v = (v = ValuedItemItem(MkValuedItem\ sterling\ (BoolVal\ true)))$

HOL Constant

$$\textbf{BoolItem} : Bool \to Item$$

$$\forall v \bullet \quad BoolItem\ v = ValuedItemItem(MkValuedItem\ sterling\ (BoolVal\ v))$$

We use the following to compute iterated conjunctions and disjunctions:

HOL Constant

$$\textbf{ListAnd} : Bool\ LIST \to Bool$$

$$\begin{array}{ll} \forall b\ bs \bullet & (ListAnd\ [] \Leftrightarrow true) \\ \wedge & (ListAnd\ (Cons\ b\ bs) \Leftrightarrow b \wedge ListAnd\ bs) \end{array}$$

HOL Constant

$$\textbf{ListOr}: Bool\ LIST \to Bool$$

$$\begin{array}{ll} \forall b\ bs \bullet & (ListOr\ [] \Leftrightarrow false) \\ \wedge & (ListOr\ (Cons\ b\ bs) \Leftrightarrow b \vee ListOr\ bs) \end{array}$$

Now, we define *BinOpAnd* which embodies the algorithm of $simplify_{ands}$. as described in [11], subject to some modifications agreed with DRA after various discussions. *BinOpAnd* itself detects whether or not the client is cleared to know the value of the expression, which permits a more lenient classification label to be computed in some useful cases.

The idea is that if any operand that the client is cleared to see is false, then the client is cleared to know that the result is false, and the classification label can be taken to be the l.u.b. of the classifications of the false operands that the client is cleared to see. If the client is not cleared to see any false operands, then the client must be cleared to see all the operands to see the result (which will be true if the client is cleared to see it) and the classification label is taken to be the l.u.b. of all the operand classifications. So that we can reuse the algorithm in *SetFuncAllAnd*, we separate out in the following function:

HOL Constant

$$\textbf{ComputeAnd} : Class \to (Class \times Item)\ LIST \to (Class \times Item)$$

$$\begin{array}{llll} \forall cc\ cil \bullet & & \multicolumn{2}{l}{ComputeAnd\ cc\ cil =} \\ & let & \multicolumn{2}{l}{hcil = cil \upharpoonright \{(c, i) \mid cc\ dominates\ c \wedge \neg ItemBool\ i\}} \\ & in\ let & \multicolumn{2}{l}{v = ListAnd(Map\ (ItemBool\ o\ Snd)\ cil)} \\ & in\ let & \multicolumn{2}{l}{makecase(c, u) = if\ \ ItemBool\ u\ \ then\ \ lattice\_bottom\ \ else\ \ c} \\ & in & if & hcil = [] \\ & & then & (lubl(Map\ Fst\ cil),\ BoolItem\ v) \\ & & else & (lubl(Map\ makecase\ hcil),\ BoolItem\ false) \end{array}$$

HOL Constant

$$\textbf{BinOpAnd} : Class \to VALUE\_COMP\ LIST \to VALUE\_COMP$$

$$\forall cc\ el \bullet BinOpAnd\ cc\ el = \\ \lambda tl\ rl\ r \bullet \\ ComputeAnd\ cc\ (Map\ (\lambda e \bullet e\ tl\ rl\ r)\ el)$$

Now *BinOpOr* which embodies the algorithm of $simplify_{ors}$, again subject to some agreed modifications. The algorithm is "dual" to that of *BinOpAnd*.

HOL Constant

$$\textbf{ComputeOr} : Class \to (Class \times Item)\ LIST \to (Class \times Item)$$

$$\begin{array}{lll} \forall cc\ cil \bullet & ComputeOr\ cc\ cil = & \\ let & hcil = cil \restriction \{(c, i) \mid cc\ dominates\ c \land ItemBool\ i\} & \\ in\ let & v = ListOr(Map\ (ItemBool\ o\ Snd)\ cil) & \\ in\ let & makecase(c, u) = if\ \ \neg ItemBool\ u\ \ then\ \ lattice\_bottom\ \ else\ \ c & \\ in & if \quad hcil = [] & \\ & then \quad (lubl(Map\ Fst\ cil), BoolItem\ v) & \\ & else \quad (lubl(Map\ makecase\ hcil), BoolItem\ true) & \end{array}$$

HOL Constant

$$\textbf{BinOpOr} : Class \to VALUE\_COMP\ LIST \to VALUE\_COMP$$

$$\forall cc\ el \bullet BinOpOr\ cc\ el = \\ \lambda tl\ rl\ r \bullet \\ ComputeOr\ cc\ (Map\ (\lambda e \bullet e\ tl\ rl\ r)\ el)$$

The other binary forms use an unoptimised treatment of classifications and are parameterised by the actual item computation to perform and expressions to give the operands.

HOL Constant

$$\textbf{BinOp}: (Item \to Item \to Item) \\ \to VALUE\_COMP \to VALUE\_COMP \to VALUE\_COMP$$

$$\begin{array}{ll} \forall f\ e1\ e2 \bullet & \\ BinOp\ f\ e1\ e2 = & \\ \lambda tl\ rl\ r \bullet & \\ let & (c1, v1) = e1\ tl\ rl\ r \\ in\ let & (c2, v2) = e2\ tl\ rl\ r \\ in & (c1\ lub\ c2, f\ v1\ v2) \end{array}$$

4.4 Triadic

The triadic forms all use an unoptimised treatment of classifications and are parameterised by the actual item computation to perform and three expressions giving the operands.

HOL Constant

$$\mathbf{TriOp} : (Item \to Item \to Item \to Item) \to VALUE\_COMP \to VALUE\_COMP \to VALUE\_COMP \to VALUE\_COMP$$

$$\begin{array}{l}
\forall f\ e1\ e2\ e3 \bullet \\
\quad TriOp\ f\ e1\ e2\ e3 = \\
\quad \lambda tl\ rl\ r \bullet \\
\quad let \quad (c1,\ v1) = e1\ tl\ rl\ r \\
\quad in\ let\ (c2,\ v2) = e2\ tl\ rl\ r \\
\quad in\ let\ (c3,\ v3) = e3\ tl\ rl\ r \\
\quad in \quad (c1\ lub\ (c2\ lub\ c3),\ f\ v1\ v2\ v3)
\end{array}$$

4.5 Conversions

See section 3.5.

4.6 Sterling

See sections 3.5 and 4.16.

4.7 Dinary

See sections 3.5 and 4.16.

4.8 Declaration

Since we are constructing a model of the semantics of expressions rather than the syntax, we do not model declarations.

4.9 Case Expressions

The class computation for case expressions is a little complicated. The idea is that evaluation of a case expression gives some information about all of the test conditions up to and including the one (if any) corresponding to the branch which is actually taken; if the client is not cleared to evaluate one of these test conditions, then the class of the whole expression is taken to be the class of the first such condition (which ensures that the client will not be cleared to see the result), otherwise the class of the whole expression is taken to be the class of the expression in the branch which is taken.

The two sorts of case expression are parameterised by the client clearance, and expressions and lists of pairs of expressions giving the operands. Lists of pairs are used rather than pairs of lists as in [11] to avoid the anomalous case where the two lists have different lengths (which is prohibited by the syntactic rules for TSQL in the context of [11]).

The following two auxiliary functions are used to compute the class for the *caseVal* form. They correspond to the queries *check_list* and *check_test* computed by $internal\_value_{class}$ in [11] (with the work performed by the *limb* functions expanded out).

HOL Constant

CheckList : $Class \to Item \to ((Class \times Item) \times (Class \times Item))\ LIST \to Class \to Class$

$$
\begin{array}{l}
\forall cc\ ti\ cv\ cvs\ elsec \bullet \\
\quad CheckList\ cc\ ti\ []\ elsec = elsec \\
\land \quad CheckList\ cc\ ti\ (Cons\ cv\ cvs)\ elsec = \\
\quad let\ ((cec, cei), (vec, vei)) = cv \\
\quad in \quad if \quad \neg cc\ dominates\ cec \\
\qquad\qquad then \quad cec \\
\qquad\qquad else\ if\ ti = cei \\
\qquad\qquad then \quad vec \\
\qquad\qquad else \quad CheckList\ cc\ ti\ cvs\ elsec
\end{array}
$$

HOL Constant

CheckTest : $Class \to (Class \times Item) \to ((Class \times Item) \times (Class \times Item))\ LIST \to Class \to Class$

$$
\begin{array}{l}
\forall cc\ ti\ tc\ cvs\ elsec \bullet \\
\quad CheckTest\ cc\ (tc, ti)\ cvs\ elsec = \\
\quad if \quad cc\ dominates\ tc \\
\quad then \quad CheckList\ cc\ ti\ cvs\ elsec \\
\quad else \quad tc
\end{array}
$$

The following function computes the value returned by the *caseVal* form of case expression.

HOL Constant

$$\textbf{CaseValValue} : Item \to ((Class \times Item) \times (Class \times Item))\ LIST \to Item \to Item$$

$$\forall ti\ cv\ cvs\ elsev \bullet$$
$$\quad CaseValValue\ ti\ []\ elsev = elsev$$
$$\wedge \quad CaseValValue\ ti\ (Cons\ cv\ cvs)\ elsev =$$
$$\quad let\ ((cec,\ cei),\ (vec,\ vei)) = cv$$
$$\quad in\quad if\quad ti = cei$$
$$\quad\quad then\quad vei$$
$$\quad\quad else\quad CaseValValue\ ti\ cvs\ elsev$$

HOL Constant

$$\textbf{CaseVal} : Class \to VALUE\_COMP \to (VALUE\_COMP \times VALUE\_COMP)\ LIST \to VALUE\_COMP \to VALUE\_COMP$$

$$\forall cc\ tst\ casevals\ elseval \bullet$$
$$\quad CaseVal\ cc\ tst\ casevals\ elseval =$$
$$\quad \lambda tl\ rl\ r \bullet$$
$$\quad let\quad (tc,\ ti) = tst\ tl\ rl\ r$$
$$\quad in\ let\quad cvs = Map\ (\lambda (c,\ v) \bullet\ (c\ tl\ rl\ r,\ v\ tl\ rl\ r))\ casevals$$
$$\quad in\ let\quad (ec,\ ei) = elseval\ tl\ rl\ r$$
$$\quad in\ let\quad c = CheckTest\ cc\ (tc,\ ti)\ cvs\ ec$$
$$\quad in\ let\quad v = CaseValValue\ ti\ cvs\ ei$$
$$\quad in\quad (c,\ v)$$

The following auxiliary functions is used to compute the class for the *case* form. It corresponds to the calculations performed by the query c in the relevant clause of $internal\_value_{class}$ in [11] (with the work performed by the *limb* functions expanded out).

HOL Constant

$$
\begin{array}{l}
\textbf{CaseC}: \mathit{Class} \to \\
\quad ((\mathit{Class} \times \mathit{Item}) \times (\mathit{Class} \times \mathit{Item}))\ \mathit{LIST} \\
\quad \to \mathit{Class} \to \mathit{Class}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall cc\ cv\ cvs\ elsec \bullet} \\
& \multicolumn{2}{l}{CaseC\ cc\ []\ elsec = elsec} \\
\land & \multicolumn{2}{l}{CaseC\ cc\ (Cons\ cv\ cvs)\ elsec =} \\
& \multicolumn{2}{l}{let\ ((cec,\ cei),\ (vec,\ vei)) = cv} \\
& in & if\quad \neg cc\ dominates\ cec \\
& & then\quad cec \\
& & else\ if\ ItemBool\ cei \\
& & then\quad vec \\
& & else\quad CaseC\ cc\ cvs\ elsec
\end{array}
$$

The following function computes the value returned by the *case* form of case expression.

HOL Constant

$$
\begin{array}{l}
\textbf{CaseValue} : ((\mathit{Class} \times \mathit{Item}) \times (\mathit{Class} \times \mathit{Item}))\ \mathit{LIST} \\
\quad \to \mathit{Item} \to \mathit{Item}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall cv\ cvs\ elsev \bullet} \\
& \multicolumn{2}{l}{CaseValue\ []\ elsev = elsev} \\
\land & \multicolumn{2}{l}{CaseValue\ (Cons\ cv\ cvs)\ elsev =} \\
& \multicolumn{2}{l}{let\ ((cec,\ cei),\ (vec,\ vei)) = cv} \\
& in & if\quad ItemBool\ cei \\
& & then\quad vei \\
& & else\quad CaseValue\ cvs\ elsev
\end{array}
$$

HOL Constant

$$
\begin{array}{l}
\textbf{Case} : \mathit{Class} \to (\mathit{VALUE\_COMP} \times \mathit{VALUE\_COMP})\ \mathit{LIST} \\
\quad \to \mathit{VALUE\_COMP} \to \mathit{VALUE\_COMP}
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\forall cc\ casevals\ elseval \bullet} \\
& \multicolumn{2}{l}{Case\ cc\ casevals\ elseval =} \\
& \multicolumn{2}{l}{\lambda tl\ rl\ r \bullet} \\
& let & cvs = Map\ (\lambda(c,\ v) \bullet\ (c\ tl\ rl\ r,\ v\ tl\ rl\ r))\ casevals \\
& in\ let & (ec,\ ei) = elseval\ tl\ rl\ r \\
& in\ let & c = CaseC\ cc\ cvs\ ec \\
& in\ let & v = CaseValue\ cvs\ ei \\
& in & (c,\ v)
\end{array}
$$

4.10 Set Functions

The logical set functions are treated in a very similar fashion to the logical binary operators.

HOL Constant

SetFuncAllAnd : $Class \to VALUE\_COMP \to VALUE\_COMP$

$\forall cc\ e \bullet SetFuncAllAnd\ cc\ e =$
$\quad \lambda tl\ rl\ r \bullet$
$\quad ComputeAnd\ cc\ (Map\ (e\ tl\ rl)\ rl)$

HOL Constant

SetFuncAllOr : $Class \to VALUE\_COMP \to VALUE\_COMP$

$\forall cc\ e \bullet SetFuncAllOr\ cc\ e =$
$\quad \lambda tl\ rl\ r \bullet$
$\quad ComputeOr\ cc\ (Map\ (e\ tl\ rl)\ rl)$

The general set functions are parameterised by the operation on lists or sets of item pairs to be computed and by the expression to be computed for each row.

HOL Constant

SetFuncAll : $(Item\ LIST \to Item) \to VALUE\_COMP \to VALUE\_COMP$

$\forall f\ e \bullet SetFuncAll\ f\ e =$
$\quad \lambda tl\ rl\ r \bullet$
$\quad let \quad (cl,\ il) = Split\ (Map\ (e\ tl\ rl)\ rl)$
$\quad in \quad (lubl\ cl,\ f\ il)$

HOL Constant

SetFuncDistinct : $(Item\ SET \to Item) \to VALUE\_COMP \to VALUE\_COMP$

$\forall f\ e \bullet SetFuncDistinct\ f\ e =$
$\quad \lambda tl\ rl\ r \bullet$
$\quad let \quad (cl,\ il) = Split\ (Map\ (e\ tl\ rl)\ rl)$
$\quad in \quad (lubl\ cl,\ f\ (Elems\ il))$

The "distinct" option of the logical set functions would appear to be semantically identical with the "all" option and so has not been given here.

4.11 Count Functions

For simplicity, we ignore the numeric type prescription which is associated with the count functions.

Examination of [11] reveals that the first two count functions may be treated as instances of the general set functions for present purposes, although to do this we need the function which converts an HOL natural number into a TSQL item, the precise details being unimportant:

HOL Constant

$$\textbf{NatItem} : \mathbb{N} \to Item$$

$$true$$

HOL Constant

$$\textbf{CountNonNull} : VALUE\_COMP \to VALUE\_COMP$$

$$\begin{array}{ll} \forall e \bullet & CountNonNull\ e = \\ & let \quad counter\ il = NatItem(Length\ (il \restriction \{i \mid isValuedItem\ i\})) \\ & in \quad SetFuncAll\ counter\ e \end{array}$$

HOL Constant

$$\textbf{CountDistinct} : VALUE\_COMP \to VALUE\_COMP$$

$$\begin{array}{ll} \forall e \bullet & CountDistinct\ e = \\ & let \quad counter\ is = NatItem(Size\ (is \cap \{i \mid isValuedItem\ i\})) \\ & in \quad SetFuncDistinct\ counter\ e \end{array}$$

The *count_all* function uses a somewhat different classification computation taking into account row existence classes.

HOL Constant

$$\textbf{CountAll} : VALUE\_COMP$$

$$\begin{array}{l} CountAll = \\ \lambda tl\ rl\ r \bullet \\ let \quad cl = Map\ DTR\_row\ rl \\ in \quad (lubl\ cl,\ NatItem(Length\ rl)) \end{array}$$

4.12 *AllBinOp*

See section 3.5.

4.13 *SomeBinOp*

See section 3.5.

4.14 *ExistsTuples*

This construct is parameterised by a table expression to compute the operand. The client clearance is also needed to filter out rows whose existence the client is not cleared to know. The clearance calculation below is intended to be in the spirit of $tuple\_list_{max\_row\_class}$ from [11].

HOL Constant

$$\textbf{ExistsTuples} : Class \to TABLE\_COMP \to VALUE\_COMP$$

$$
\begin{array}{l}
\forall cc\ te \bullet \\
\quad ExistsTuples\ cc\ te = \\
\quad \lambda tl\ rl\ r \bullet \\
\quad let \quad (c,\ t) = te\ tl \\
\quad in\ let \quad trl = DT\_rows\ t \restriction \\
\qquad\qquad \{r \mid cc\ dominates\ DTR\_row\ r \land cc\ dominates\ DTR\_where\ r\} \\
\quad in \quad if \quad cc\ dominates\ c \\
\qquad then \quad (lubl\ (Map\ DTR\_row\ trl),\ BoolItem\ (\neg trl = [])) \\
\qquad else \quad (c,\ Arbitrary)
\end{array}
$$

(Above used *lattice_top* instead of the very last c in an earlier version discussed on the 'phone with DRA. c is probably still not the ideal label for information purposes.)

4.15 *SingleValue*

This construct is parameterised by a table expression to compute the operand and the client clearance (to allow rows whose existence the client is not cleared to see to be hidden). The classification below is sometimes more generous than that in $internal\_value_{class}$ from [11] (which uses the statically known upper bound on the class of the single column in the table).

HOL Constant

$$\textbf{SingleValue} : Class \to TABLE\_COMP \to VALUE\_COMP$$

$$\begin{array}{l}
\forall cc\ te \bullet \\
\quad SingleValue\ cc\ te = \\
\quad \lambda tl\ rl\ r \bullet \\
\quad let \quad (c,\ t) = te\ tl \\
\quad in\ let \quad trl = DT_rows\ t \upharpoonright \\
\qquad\qquad \{r \mid cc\ dominates\ DTR_row\ r \land cc\ dominates\ DTR_where\ r\} \\
\quad in\ let \quad cil = DTR_cols\ (Hd\ trl) \\
\quad in\ let \quad (ic,\ ii) = Hd\ cil \\
\quad in \quad if \quad cc\ dominates\ c \\
\qquad then \quad if\ Length\ trl = 1 \land Length\ cil = 1 \\
\qquad\qquad then \quad (ic,\ ii) \\
\qquad\qquad else \quad (c,\ Arbitrary) \\
\qquad else \quad (c,\ Arbitrary)
\end{array}$$

(As with *ExistsTuples* the information label could probably do better here.)

4.16 Contents

Since we have not upgraded to the new SSQL specification we only supply one contents operator rather than separate sterling and dinary ones.

The contents operator is parameterised by a number telling us which column to select. We must return a fixed value if this is out of range (a situation which never arises in the actual SWORD implementation since the column is identified by name rather than number and it is known at compile/transformation time whether or not the name is valid).

HOL Constant

$$\textbf{Contents} : \mathbb{N} \to VALUE\_COMP$$

$$\begin{array}{l}
\forall i \bullet \quad Contents\ i = \\
\quad \lambda tl\ rl\ r \bullet \\
\quad if \quad 1 \le i \land i \le Length\ (DTR_cols\ r) \\
\quad then \quad Nth\ (DTR_cols\ r)\ i \\
\quad else \quad Arbitrary
\end{array}$$

4.17 *Classification*

This construct is parameterised by a number telling us the column whose classification is to be revealed. The classification below is as in $internal\_value_{class}$ from [11].

HOL Constant

$$\textbf{ClassItem} : \mathit{Class} \to \mathit{Item}$$

$$\forall v \bullet \quad \mathit{ClassItem}\ v = \mathit{ValuedItemItem}(\mathit{MkValuedItem}\ \mathit{sterling}\ (\mathit{ClassVal}\ v))$$

HOL Constant

$$\textbf{Classification} : \mathbb{N} \to \mathit{VALUE\_COMP}$$

$$\begin{aligned}
\forall i \bullet \quad & \mathit{Classification}\ i = \\
& \lambda tl\ rl\ r \bullet \\
& \mathit{if} \quad 1 \le i \land i \le \mathit{Length}\ (\mathit{DTR_cols}\ r) \\
& \mathit{then} \quad (\mathit{DTR_row}\ r,\ \mathit{ClassItem}(\mathit{Fst}\ (\mathit{Nth}\ (\mathit{DTR_cols}\ r)\ i))) \\
& \mathit{else} \quad \mathit{Arbitrary}
\end{aligned}$$

4.18 *RowExistence*

See section 3.5.

4.19 *JoinedRowExistence*

This just returns the row existence field of the current row. For simplicity, it uses the client clearance, passed as a parameter, as the result clearance. This is stronger than [11] in general.

HOL Constant

$$\textbf{JoinedRowExistence} : \mathit{Class} \to \mathit{VALUE\_COMP}$$

$$\begin{aligned}
\forall cc \bullet \quad & \mathit{JoinedRowExistence}\ cc = \\
& \lambda tl\ rl\ r \bullet (cc,\ \mathit{ClassItem}(\mathit{DTR_row}\ r))
\end{aligned}$$

5 TABLE COMPUTATIONS

To specify the *SELECT* and related queries it is convenient to break them down into more primitive operations: *join*, *projection* etc. In this section, we give these top-level building blocks and then combine them in various ways to describe the *SELECT* query and its variants.

5.1 Auxiliary Functions

5.1.1 Join

The TSQL join operation is just cartesian product. The row class and where clause class in each tuple in the product is the least upper bound of the corresponding classes in each component tuple which contributes to that tuple. The joined table has no name and the maximum row class is the least upper bound of those in the component tables.

Cf. $tuple\_list_{make\_outer}$ in [11].

HOL Constant

JoinSpecs : *DerTableSpec LIST* → *DerTableSpec*

$\forall sl \bullet$
$JoinSpecs\ sl =$
$let \quad n = []$
$and \quad mr = lubl\ (Map\ DTS\_maxRow\ sl)$
$and \quad csl = Flat\ (Map\ DTS\_colSpecs\ sl)$
$in \quad MkDerTableSpec\ n\ mr\ csl$

HOL Constant

JoinRows : *DerTableRow* → *DerTableRow LIST* → *DerTableRow LIST*

$\forall r\ rs \bullet$
$JoinRows\ r\ rs =$
$let \quad join2\ rr = ($
$\qquad MkDerTableRow$
$\qquad (DTR\_where\ r\ lub\ DTR\_where\ rr)$
$\qquad (DTR\_row\ r\ lub\ DTR\_row\ rr)$
$\qquad (DTR\_cols\ r \frown DTR\_cols\ rr))$
$in \quad Map\ join2\ rs$

HOL Constant

$$\mathbf{JoinData} : DerTableRow\ LIST\ LIST \to DerTableRow\ LIST$$

$$\begin{array}{ll}
& JoinData\ [] = [] \\
\land & \forall tab\ rest \bullet \\
& JoinData\ (Cons\ tab\ rest) = \\
& \mathit{if}\quad rest = [] \\
& \mathit{then}\quad tab \\
& \mathit{else}\quad let\quad jrest = JoinData\ rest \\
& \qquad in\ let\quad join_blk\ r = JoinRows\ r\ jrest \\
& \qquad in\quad Flat\ (Map\ join_blk\ tab)
\end{array}$$

The join operation itself is parameterised by the table expressions which compute the tables to be joined.

HOL Constant

$$\mathbf{Join} : DerTable\ LIST \to (DerTableSpec \times DerTableRow\ LIST)$$

$$\begin{array}{l}
\forall tabl \bullet\ Join\ tabl = \\
\quad (JoinSpecs\ (Map\ DT_spec\ tabl),\ JoinData\ (Map\ DT_rows\ tabl))
\end{array}$$

5.1.2 Projection

The projection operation is parameterised by a list of functions which compute class-item pairs from a row of a table, together with column specifications to use for the computed fields. (Note this is a more general than projection of a single field to form a one-column table but includes that as a special case). The operation is required in two flavours, one to support *all_tuples* and *distinct_tuples* and one to support *evaluate*.

In the SWORD implementation the column specifications are effectively computed during the transformations as required. In the formalisation here, the column specifications are not in fact significant, but have been left in to allow for possible further development to model some of the optimisations performed by the transformations.

The operation acts on a table computed by a table expression given as a parameter. The parameter expressions are expected to assign appropriate classifications to the resulting fields. The computed table is anonymous.

Cf. $tuple\_list_{make\_outer}$ in [11].

HOL Constant

$$\textbf{ProjectSpec} : DerColSpec\ LIST \to DerTableSpec \to DerTableSpec$$

$$\forall sl\ s \bullet\ ProjectSpec\ sl\ s = MkDerTableSpec\ []\ (DTS\_maxRow\ s)\ sl$$

HOL Constant

$$\textbf{ProjectData} : DerTable\ LIST \to VALUE\_COMP\ LIST \to DerTableRow\ LIST\ LIST \to DerTableRow\ LIST$$

$$\begin{array}{l}\forall tl\ el\ gps \bullet \\ \quad ProjectData\ tl\ el\ gps = \\ \quad let\quad h\ gp\ r = \quad MkDerTableRow \\ \quad\quad\quad (DTR\_where\ r)\ (DTR\_row\ r)\ (Map\ (\lambda e \bullet e\ tl\ gp\ r)\ el) \\ \quad in\ let\quad k\ gp = Map\ (h\ gp)\ gp \\ \quad in\quad Flat\ (Map\ k\ gps)\end{array}$$

HOL Constant

$$\textbf{Project} : DerTableSpec \to DerTable\ LIST \to (VALUE\_COMP \times DerColSpec)\ LIST \to DerTableRow\ LIST\ LIST \to DerTable$$

$$\begin{array}{l}\forall ts\ tl\ sellist\ gps \bullet \\ \quad Project\ ts\ tl\ sellist\ gps = \\ \quad MkDerTable \\ \quad (ProjectSpec\ (Map\ Snd\ sellist)\ ts) \\ \quad (ProjectData\ tl\ (Map\ Fst\ sellist)\ gps)\end{array}$$

HOL Constant

$$\begin{array}{l}
\textbf{EvalProjectData} : DerTable\ LIST \\
\quad \to VALUE_COMP\ LIST \\
\quad \to DerTableRow\ LIST\ LIST \\
\quad \to DerTableRow\ LIST
\end{array}$$

$$\begin{array}{l}
\forall tl\ el\ gps \bullet \\
\quad EvalProjectData\ tl\ el\ gps = \\
\quad let \quad h\ gp\ r = \quad MkDerTableRow \\
\qquad\qquad (DTR_where\ r)\ (DTR_row\ r)\ (Map\ (\lambda e \bullet e\ tl\ gp\ r)\ el) \\
\quad in\ let \quad k\ gp = \\
\qquad let \quad results = Map\ (h\ gp)\ gp \\
\qquad in \quad if \quad Size(Elems\ results) = 1 \\
\qquad\qquad then \quad Hd\ results \\
\qquad\qquad else \quad Arbitrary \\
\quad in \quad Map\ k\ gps
\end{array}$$

HOL Constant

$$\begin{array}{l}
\textbf{EvalProject} : DerTableSpec \\
\quad \to DerTable\ LIST \\
\quad \to (VALUE_COMP \times DerColSpec)\ LIST \\
\quad \to DerTableRow\ LIST\ LIST \\
\quad \to DerTable
\end{array}$$

$$\begin{array}{l}
\forall ts\ tl\ sellist\ gps \bullet \\
\quad EvalProject\ ts\ tl\ sellist\ gps = \\
\quad MkDerTable \\
\quad (ProjectSpec\ (Map\ Snd\ sellist)\ ts) \\
\quad (EvalProjectData\ tl\ (Map\ Fst\ sellist)\ gps)
\end{array}$$

5.1.3 *WHERE*

The *WHERE* operation is parameterised, amongst other things, by a classification to be used to eliminate rows whose existence the client is not allowed to know. The classification computations follow $tuple\_list_{make\_outer}$ in [11]. The local function w computes the class of the where clauses for each new row. The local function h computes for each row a pair comprising a truth value, indicating whether the new row is wanted, and a row, being the row to appear in the result if the row is wanted.

HOL Constant

$$\textbf{Where} : Class \to DerTable\ LIST \to DerTableRow\ LIST \to VALUE\_COMP \to DerTableRow\ LIST$$

$$\begin{array}{l} \forall c\ tl\ rl\ e \bullet \\ \quad Where\ c\ tl\ rl\ e = \\ \quad let \quad hrl = rl \restriction \{r \mid c\ dominates\ DTR\_row\ r\} \\ \quad in\ let \quad w\ r = (DTR\_where\ r\ lub\ Fst\ (e\ tl\ hrl\ r)) \\ \quad in\ let \quad h\ r = ((ItemBool(Snd\ (e\ tl\ hrl\ r)) \lor \neg\ c\ dominates\ w\ r), \\ \quad\quad\quad MkDerTableRow\ (w\ r)\ (DTR\_row\ r)\ (DTR\_cols\ r)) \\ \quad in \quad Map\ Snd\ (Map\ h\ hrl \restriction \{(t,\ r) \mid t\}) \end{array}$$

5.1.4 *GROUPBY/HAVING*

The *GROUPBY* and *HAVING* operations must be treated together. It is here that the classification produced by a *TABLE_COMP* is computed in anger.

Cf. $tuple\_list_{make\_outer}$ in [11].

To define the grouping part of the operation, we need some list processing preliminaries. The next two functions are parameterised by a function *gpby* which is used to decide whether two rows are in the same group. The idea is that two rows, x and y are in the same group if $gpby\, x = gpby\, y$. Given a row and a partial list of groups, *PutInGroup* adds the row to the appropriate group in the list (creating a new group with only one row, if necessary).

HOL Constant

$$\textbf{PutInGroup} : ('a \to 'b) \to 'a \to ('a\ LIST)\ LIST \to ('a\ LIST)\ LIST$$

$$\begin{array}{l} \forall gpby\ x\ gp\ gps \bullet \\ \quad PutInGroup\ gpby\ x\ [] = [[x]] \\ \land \quad PutInGroup\ gpby\ x\ (Cons\ gp\ gps) = \\ \quad if \quad gpby\ x = gpby\ (Hd\ gp) \\ \quad then \quad Cons\ (Cons\ x\ gp)\ gps \\ \quad else \quad Cons\ gp\ (PutInGroup\ gpby\ x\ gps) \end{array}$$

MakeGroups uses *PutInGroup* to do the complete job of organising a list into groups according to a given *gpby* function.

HOL Constant

$$\textbf{MakeGroups} : ('a \to 'b) \to 'a\ LIST \to ('a\ LIST)\ LIST$$

$$\begin{array}{ll} \forall gpby\ x\ xs \bullet & \\ & MakeGroups\ gpby\ [] = [] \\ \land & MakeGroups\ gpby\ (Cons\ x\ xs) = PutInGroup\ gpby\ x\ (MakeGroups\ gpby\ xs) \end{array}$$

The following function is used to supply the *gpby* parameter to the above. It ensures that indices which are out of range are mapped to a fixed arbitrary value to handle securely an error condition which is detected at compile/transformation time in the SWORD implementation (where names rather than numbers are used for the columns and the scope rules detect invalid names).

HOL Constant

$$\textbf{ListNth} : \mathbb{N}\ LIST \to 'a\ LIST \to 'a\ LIST$$

$$\begin{array}{lll} \forall n\ nl\ list \bullet & & \\ & ListNth\ []\ list = [] & \\ \land & ListNth\ (Cons\ n\ nl)\ list = & \\ & Cons & \\ & (if & 1 \le n \land n \le Length\ list \\ & then & Nth\ list\ n \\ & else & Arbitrary) \\ & (ListNth\ nl\ list) & \end{array}$$

The following is how we analyse the result of the expressions which appear in *HAVING* clauses which take the contents of the columns which appear in the *GROUPBY* clause. If the (item part of the) expression parameter gives the same value in each row then that value is returned classified with the l.u.b. of the classes returned for the rows; otherwise, a fixed arbitrary value is returned with the same classification (modelling a run-time error). This effect may be achieved using the general set function *SetFuncAll* as follows:

HOL Constant

$$\textbf{CommonValue} : VALUE\_COMP \to VALUE\_COMP$$

$$\begin{array}{llll} \forall e \bullet & CommonValue\ e = & & \\ & let & pick\ il = & \\ & & if & Size\ (Elems\ il) = 1 \\ & & then & Hd\ il \\ & & else & Arbitrary \\ & in & SetFuncAll\ pick\ e & \end{array}$$

The classification computation below is a suggestion on the basis of current information. If the client is not cleared to compute the grouping, then the result class is the l.u.b. of the classes in the columns required to do the grouping (which the client will not dominate). If the client is cleared to compute the grouping, then the result class is the l.u.b. of the classes of the results of all the *HAVING* tests. If the client clearance does not dominate this result class then the optional check query generated by $tuple\_list_{make\_outer}$ would return some rows (i.e., the check would fail and the query would not be allowed to proceed).

Note that when *Group* is used, rows whose existence the client is not cleared to know have been filtered out (using *Where*), and consequently no information flows arise from the grouping on class columns does not contribute to the result class (cf. $tuple\_list_{make\_outer}$).

HOL Constant

$$
\begin{array}{ll}
\textbf{Group}: & Class \\
& \rightarrow DerTable\ LIST \\
& \rightarrow DerTableRow\ LIST \\
& \rightarrow \mathbb{N}\ LIST \\
& \rightarrow \mathbb{N}\ LIST \\
& \rightarrow VALUE\_COMP \\
& \rightarrow (Class \times (DerTableRow\ LIST\ LIST))
\end{array}
$$

$$
\begin{array}{l}
\forall cc\ tl\ rl\ gbsterling\ gbclass\ having \bullet \\
\quad Group\ cc\ tl\ rl\ gbsterling\ gbclass\ having = \\
\quad let \quad gpby\ row = (ListNth\ gbsterling\ (Map\ Snd\ (DTR\_cols\ row)), \\
\quad \qquad\qquad ListNth\ gbclass\ (Map\ Fst(DTR\_cols\ row))) \\
\quad in\ let\ \ gbc\ row = lubl(ListNth\ gbsterling\ (Map\ Fst(DTR\_cols\ row))) \\
\quad in\ let\ \ gps = MakeGroups\ gpby\ rl \\
\quad in\ let\ \ has\_test\ gp = ((CommonValue\ having)\ tl\ gp\ Arbitrary) \\
\quad in\ let\ \ cl = \ if \quad cc\ dominates\ lubl\ (Map\ gbc\ rl) \\
\quad \qquad\qquad then \quad lubl\ (Map\ (Fst\ o\ has\_test)\ gps) \\
\quad \qquad\qquad else \quad lubl\ (Map\ gbc\ rl) \\
\quad in\ let\ \ wanted\_gps = gps \upharpoonright \{gp \mid ItemBool(Snd\ (has\_test\ gp))\} \\
\quad in \quad (cl,\ wanted\_gps)
\end{array}
$$

5.2 *TableContents*

This is parameterised by a number giving the index into the list of tables. The class component of the result is bottom, indicating that no grouping has yet been done.

HOL Constant

$$\textbf{TableContents} \quad : \mathbb{N} \to TABLE\_COMP$$

$$\begin{array}{l}
\forall i \bullet \\
\quad TableContents\ i = \\
\quad \lambda tl \bullet \\
\quad if \quad 1 \leq i \land i \leq Length\ tl \\
\quad then \quad (lattice_bottom,\ Nth\ tl\ i) \\
\quad else \quad Arbitrary
\end{array}$$

5.3 *AllTuples*

HOL Constant

$$\begin{array}{ll}
\textbf{AllTuples} & : Class \\
& \to (VALUE_COMP \times DerColSpec)\ LIST \\
& \to TABLE_COMP\ LIST \\
& \to VALUE_COMP \\
& \to \mathbb{N}\ LIST \\
& \to \mathbb{N}\ LIST \\
& \to VALUE_COMP \\
& \to TABLE_COMP
\end{array}$$

$$\begin{array}{l}
\forall cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having \bullet \\
\quad AllTuples\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having = \\
\quad \lambda tl \bullet \\
\quad let \quad (cll,\ tabs) = Split(Map\ (\lambda te \bullet te\ tl)\ fromspec) \\
\quad in\ let \quad (ts,\ tab1) = Join\ tabs \\
\quad in\ let \quad tab2 = Where\ cc\ tl\ tab1\ where \\
\quad in\ let \quad (cl1,\ gps) = Group\ cc\ tl\ tab2\ gbsterling\ gbclass\ having \\
\quad in\ let \quad cl2 = \begin{array}{ll} if & cc\ dominates\ lubl\ cll \\ then & cl1 \\ else & lubl\ cll \end{array} \\
\quad in \quad (cl2,\ Project\ ts\ tl\ sellist\ gps)
\end{array}$$

5.4 *DistinctTuples*

This is very similar to *AllTuples*; however we need a list-processing auxiliary to remove duplicate items from a list.

HOL Constant

$$\mathbf{RemoveDuplicates} : 'a\ LIST \to 'a\ LIST$$

$$\begin{aligned}
&\forall x\ xs \bullet \\
&\quad RemoveDuplicates\ [] = [] \\
\wedge &\quad RemoveDuplicates\ (Cons\ x\ xs) \\
&\quad = Cons\ x\ (RemoveDuplicates\ xs \upharpoonright \{y \mid \neg y = x\})
\end{aligned}$$

HOL Constant

$$\begin{aligned}
\mathbf{DistinctTuples} \quad : &\ Class \\
&\to (VALUE_COMP \times DerColSpec)\ LIST \\
&\to TABLE_COMP\ LIST \\
&\to VALUE_COMP \\
&\to \mathbb{N}\ LIST \\
&\to \mathbb{N}\ LIST \\
&\to VALUE_COMP \\
&\to TABLE_COMP
\end{aligned}$$

$$\begin{aligned}
&\forall cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having \bullet \\
&\quad DistinctTuples\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having = \\
&\quad \lambda tl \bullet \\
&\quad let \quad (cll,\ tabs) = Split(Map\ (\lambda te \bullet te\ tl)\ fromspec) \\
&\quad in\ let \quad (ts,\ tab1) = Join\ tabs \\
&\quad in\ let \quad tab2 = Where\ cc\ tl\ tab1\ where \\
&\quad in\ let \quad (cl,\ gps) = Group\ cc\ tl\ tab2\ gbsterling\ gbclass\ having \\
&\quad in\ let \quad rem_dups\ tab = MkDerTable(DT_spec\ tab)(RemoveDuplicates(DT_rows\ tab)) \\
&\quad in \quad (cl\ lub\ lubl\ cll,\ rem_dups(Project\ ts\ tl\ sellist\ gps))
\end{aligned}$$

N.B. the above is known not to be secure (see section 3.5) and its uses have been commented out later on.

5.5 *Evaluate*

Like *DistinctTuples* this is very similar to *AllTuples*

HOL Constant

$$
\begin{array}{ll}
\textbf{Evaluate} & : Class \\
& \to (VALUE\_COMP \ \times DerColSpec)\ LIST \\
& \to TABLE\_COMP\ LIST \\
& \to VALUE\_COMP \\
& \to \mathbb{N}\ LIST \\
& \to \mathbb{N}\ LIST \\
& \to VALUE\_COMP \\
& \to TABLE\_COMP
\end{array}
$$

$$
\begin{array}{l}
\forall cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having \bullet \\
\quad Evaluate\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having = \\
\quad \lambda tl \bullet \\
\quad \begin{array}{ll}
let & (cll,\ tabs) = Split(Map\ (\lambda te \bullet te\ tl)\ fromspec) \\
in\ let & (ts,\ tab1) = Join\ tabs \\
in\ let & tab2 = Where\ cc\ tl\ tab1\ where \\
in\ let & (cl,\ gps) = Group\ cc\ tl\ tab2\ gbsterling\ gbclass\ having \\
in & (cl\ lub\ lubl\ cll,\ EvalProject\ ts\ tl\ sellist\ gps)
\end{array}
\end{array}
$$

N.B. the above is known not to be secure (see section 3.5) and its uses have been commented out later on.

6 CLOSURE OPERATION

We would now like to form a large set of allowable table computations, namely, the set of all computations which can be performed by composing the ones defined above at a given client clearance. To abbreviate the definition we use the following function which given a set of pairs of sets, (a_i, b_j), returns a pair comprising the intersection of all the a_i and the intersection of all the b_j).

HOL Constant

$$\bigcap_2 : ('a\ SET \times 'b\ SET)\ SET \rightarrow 'a\ SET \times 'b\ SET$$

$$\forall u \bullet \bigcap_2\ u = (\bigcap\{a \mid \exists\ b \bullet (a,\ b) \in u\}, \bigcap\{b \mid \exists\ a \bullet (a,\ b) \in u\})$$

Now we give the definition, which is long but straightforward in its derivation from the various semantic functions. Note that the last two clauses are commented out, since the relevant semantic functions as currently formulated above are known not to be secure.

The order of the clauses below is a rearrangement of the order in [11] to bring semantic functions with similar signatures together.

HOL Constant

TableComputations : $Class \rightarrow TABLE\_COMP\ SET$;
ValueComputations : $Class \rightarrow VALUE\_COMP\ SET$

$$\begin{array}{l}
\forall cc \bullet \\
\quad (TableComputations\ cc,\ ValueComputations\ cc) = \\
\quad \bigcap_2 \{\ (tes,\ es) \mid \\
\qquad (\forall ci \bullet\ DenoteConstant\ ci \in es) \\
\quad \wedge \quad (\forall i \bullet\ Contents\ i \in es) \\
\quad \wedge \quad (\forall i \bullet\ Classification\ i \in es) \\
\quad \wedge \quad CountAll \in es \\
\\
\quad \wedge \quad (\forall f\ e \bullet\ e \in es \Rightarrow MonOp\ f\ e \in es) \\
\quad \wedge \quad (\forall f\ e1\ e2 \bullet\ e1 \in es \wedge e2 \in es \Rightarrow BinOp\ f\ e1\ e2 \in es) \\
\quad \wedge \quad (\forall f\ e1\ e2\ e3 \bullet\ e1 \in es \wedge e2 \in es \wedge e3 \in es \Rightarrow TriOp\ f\ e1\ e2\ e3 \in es) \\
\quad \wedge \quad (\forall el \bullet\ Elems\ el \subseteq es \Rightarrow BinOpAnd\ cc\ el \in es) \\
\quad \wedge \quad (\forall el \bullet\ Elems\ el \subseteq es \Rightarrow BinOpOr\ cc\ el \in es) \\
\\
\quad \wedge \quad (\forall te\ cel\ ee \bullet\ te \in es \wedge Elems(Map\ Fst\ cel) \subseteq es\ \wedge \\
\qquad\qquad Elems(Map\ Snd\ cel) \subseteq es \wedge ee \in es \Rightarrow \\
\qquad\qquad\quad CaseVal\ cc\ te\ cel\ ee \in es) \\
\quad \wedge \quad (\forall cel\ ee \bullet\ Elems(Map\ Fst\ cel) \subseteq es\ \wedge \\
\qquad\qquad Elems(Map\ Snd\ cel) \subseteq es \wedge ee \in es \Rightarrow \\
\qquad\qquad\quad Case\ cc\ cel\ ee \in es) \\
\\
\quad \wedge \quad (\forall e \bullet\ e \in es \Rightarrow SetFuncAllAnd\ cc\ e \in es)
\end{array}$$

$$
\begin{array}{lll}
& \land & (\forall e \bullet\ e \in es \Rightarrow \mathit{SetFuncAllOr}\ cc\ e \in es) \\
& \land & (\forall e \bullet\ e \in es \Rightarrow \mathit{CountNonNull}\ e \in es) \\
& \land & (\forall e \bullet\ e \in es \Rightarrow \mathit{CountDistinct}\ e \in es) \\
& \land & (\forall e \bullet\ e \in es \Rightarrow \mathit{CommonValue}\ e \in es) \\
& & \\
& \land & (\forall f\ e \bullet\ e \in es \Rightarrow \mathit{SetFuncAll}\ f\ e \in es) \\
& \land & (\forall f\ e \bullet\ e \in es \Rightarrow \mathit{SetFuncDistinct}\ f\ e \in es) \\
& & \\
& \land & (\forall te \bullet\ te \in tes \Rightarrow \mathit{ExistsTuples}\ cc\ te \in es) \\
& \land & (\forall te \bullet\ te \in tes \Rightarrow \mathit{SingleValue}\ cc\ te \in es) \\
& & \\
& \land & (\mathit{JoinedRowExistence}\ cc \in es) \\
& & \\
& \land & (\forall i \bullet\ \mathit{TableContents}\ i \in tes) \\
& & \\
& \land & (\forall esl\ tel\ e1\ ml\ nl\ e2 \bullet \quad \mathit{Elems}(\mathit{Map}\ \mathit{Fst}\ esl) \subseteq es \\
& & \qquad \land\ \mathit{Elems}\ tel \subseteq tes \land e1 \in es \land e2 \in es \\
& & \qquad\qquad \Rightarrow \mathit{AllTuples}\ cc\ esl\ tel\ e1\ ml\ nl\ e2 \in tes) \\
(* & \land & (\forall esl\ tel\ e1\ ml\ nl\ e2 \bullet \quad \mathit{Elems}(\mathit{Map}\ \mathit{Fst}\ esl) \subseteq es \\
& & \qquad \land\ \mathit{Elems}\ tel \subseteq tes \land e1 \in es \land e2 \in es \\
& & \qquad\qquad \Rightarrow \mathit{DistinctTuples}\ cc\ esl\ tel\ e1\ ml\ nl\ e2 \in tes) \\
& \land & (\forall esl\ tel\ e1\ ml\ nl\ e2 \bullet \quad \mathit{Elems}(\mathit{Map}\ \mathit{Fst}\ esl) \subseteq es \\
& & \qquad \land\ \mathit{Elems}\ tel \subseteq tes \land e1 \in es \land e2 \in es \\
& & \qquad\qquad \Rightarrow \mathit{Evaluate}\ cc\ esl\ tel\ e1\ ml\ nl\ e2 \in tes)\ *) \\
& \} &
\end{array}
$$

7 CRITICAL PROPERTIES

We now wish to relate the above with the critical properties defined in [6] and in particular with the notion of a "risk input". Each table computation as defined in the previous section is a function, *te* say, which computes from a list of derived tables, *tl* say, a pair comprising a classification, *c*, and a new derived table, *t*. Restricting attention to the second component of the pair (and tacking on an empty list of errors) gives us a function to which the definition of a risk input in [6] applies. The intention is that *tl* will be a risk input only in cases where the client clearance does not dominate *c*. This is captured formally by the following property of table computations.

HOL Constant

$$\textbf{OkTableComputation} : \mathit{Class} \to \mathit{TABLE\_COMP}\ \mathbb{P}$$

$$
\begin{array}{l}
\forall cc\ te \bullet \\
\quad te \in \mathit{OkTableComputation}\ cc \Leftrightarrow \\
\quad \mathit{let}\quad c\ tl = \mathit{Fst}(te\ tl) \\
\quad \mathit{in\ let}\quad f\ tl = (\mathit{Snd}(te\ tl),\ []) \\
\quad \mathit{in}\quad \mathit{RiskInputs}\ cc\ f \subseteq \{tl \mid \neg cc\ \mathit{dominates}\ c\ tl\}
\end{array}
$$

We now say that an SSQL Query Transformation Processor is OK with respect to a given compiler if the queries it produces compile to functions which could be computed by an element of the set of table computations defined in the previous section and for which the presence and behaviour of the optional check query relates appropriately to the classification computed by that table computations.

HOL Constant

$$
\begin{array}{l}
\textbf{OkSTP} : (\mathit{Query} \to (\mathit{DerTable}\ \mathit{LIST} \to (\mathit{DerTable} \times \mathit{Errors}))) \\
\quad \to (\mathit{Query},\ '\mathit{PARS})\ \mathit{STP\_TYPE}\ \mathbb{P}
\end{array}
$$

$$
\begin{array}{l}
\forall compile\ stp \bullet \\
\quad stp \in \mathit{OkSTP}\ compile \Leftrightarrow \\
\quad \forall q\ c \bullet \\
\quad \mathit{isError}(stp(q,\ c)) \lor \\
\quad \mathit{let}\quad (dq,\ ocq,\ pars) = \mathit{destVal}(stp(q,\ c)) \\
\quad \mathit{in}\quad \exists dte \bullet \\
\qquad dte \in \mathit{TableComputations}\ c \\
\quad \land\quad compile\ dq = (\lambda tl \bullet (\mathit{Snd}(dte\ tl),\ [])) \\
\quad \land\quad \forall tl \bullet \\
\qquad \neg c\ \mathit{dominates}\ (\mathit{Fst}\ (dte\ tl)) \Rightarrow \\
\qquad\quad \mathit{IsL}\ ocq \\
\qquad \land\quad \mathit{is\_select}\ (\mathit{OutL}\ ocq) \\
\qquad \land\quad \neg \mathit{DT\_rows}(\mathit{Fst}(compile\ (\mathit{OutL}\ ocq)\ tl)) = []
\end{array}
$$

The above says that each output of *stp* must either indicate a transformation-time error, or must generate a data query, *dq*, which could equally well be computed by the table computation *dte*. Moreover, in the latter case, if there is any list of derived tables, *tl*, for which *dte* would return a class which is not dominated by the client clearance, then *stp* must generate a check query and the resulting check must fail on *tl*. (For simplicity, and because that is the line taken in [11], we actually insist that the check query fail by producing some rows rather than by causing an error to be signalled.)

The relationship of the notions defined in this section and the critical properties for the SWORD system as a whole is discussed and formalised in [8].

The specification of *OkTableComputation* above relates to the SWORD execution model. We define OK_TC_d as a more straightforward statement that a set of table computations does in fact exhibit the required information flow properties, viz. that if two lists of derived tables are the same when viewed by a client, and information is revealed by a particular table computation, then the client's clearance does not dominate the classification of that table computation.

HOL Constant

$$\mathbf{OK\_TC_d} : Class \rightarrow TABLE\_COMP\ \mathbb{P}$$

$$\begin{array}{l}
\forall c\ tc \bullet\ tc \in OK_TC_d\ c \Leftrightarrow \\
\quad \forall tl_0\ tl_1 \bullet \\
\quad\quad Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 \\
\quad \wedge \quad \neg HideDerTable\ c\ (Snd(tc\ tl_0)) = HideDerTable\ c\ (Snd(tc\ tl_1)) \\
\quad \Rightarrow \quad \neg c\ dominates\ Fst(tc\ tl_0)
\end{array}$$

The proof document [7] gives a formal proof that any table computation which has the OK_TC_d property also has the *OKTableComputation* property. In [9], it is proved that every table computation has the OK_TC_d property (at the appropriate classification). The proof is by a simultaneous induction over the table and value computations. The property of value computations which is proved during the course of this induction is captured by the following set (which has a natural interpretation in terms of the constraints on the flow of information into an individual data cell within a table output by the system).

HOL Constant

$$\mathbf{OK\_VC_d} : Class \rightarrow VALUE\_COMP\ \mathbb{P}$$

$$\begin{array}{l}
\forall c\ vc \bullet\ vc \in OK_VC_d\ c \Leftrightarrow \\
\quad \forall tl_0\ tl_1\ rl_0\ rl_1\ r_0\ r_1 \bullet \\
\quad\quad Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 \\
\quad \wedge \quad Map\ (HideDerTableRow\ c)\ rl_0 = Map\ (HideDerTableRow\ c)\ rl_1 \\
\quad \wedge \quad HideDerTableRow\ c\ r_0 = HideDerTableRow\ c\ r_1 \\
\quad \wedge \quad \neg Snd(vc\ tl_0\ rl_0\ r_0) = Snd(vc\ tl_1\ rl_1\ r_1) \\
\quad \Rightarrow \quad \neg c\ dominates\ Fst(vc\ tl_0\ rl_0\ r_0)
\end{array}$$

It turns out that the properties of the above sets are too weak to carry out the inductive proof. The inductive hypotheses must be strengthened. The additional hypothesis on the value computations is the intuitively necessary condition that classification labels in tables output by the system do not give rise to illicit information flow. This property is captured in the following definition:

HOL Constant

$$\mathbf{OK\_VC_c} : Class \to VALUE\_COMP\ \mathbb{P}$$

$$\begin{array}{l}
\forall c\ vc \bullet\ vc \in OK_VC_c\ c \Leftrightarrow \\
\quad \forall tl_0\ tl_1\ rl_0\ rl_1\ r_0\ r_1 \bullet \\
\qquad\quad Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 \\
\quad \land \quad Map\ (HideDerTableRow\ c)\ rl_0 = Map\ (HideDerTableRow\ c)\ rl_1 \\
\quad \land \quad HideDerTableRow\ c\ r_0 = HideDerTableRow\ c\ r_1 \\
\quad \Rightarrow \quad Fst(vc\ tl_0\ rl_0\ r_0) = Fst(vc\ tl_1\ rl_1\ r_1)
\end{array}$$

An analogous strengthening of the hypotheses on the table computations is also required (because nested queries cause a flow from the classification resulting from a table computation into that resulting from the surrounding value computation). The following definition states what is required:

HOL Constant

$$\mathbf{OK\_TC_c} : Class \to TABLE\_COMP\ \mathbb{P}$$

$$\begin{array}{l}
\forall c\ tc \bullet\ tc \in OK_TC_c\ c \Leftrightarrow \\
\quad \forall tl_0\ tl_1 \bullet \\
\qquad\quad Map\ (HideDerTable\ c)\ tl_0 = Map\ (HideDerTable\ c)\ tl_1 \\
\quad \Rightarrow \quad Fst(tc\ tl_0) = Fst(tc\ tl_1)
\end{array}$$

8 CLOSING DOWN

The following ProofPower instruction restores the previous proof context.

SML

```
pop_pc();
```

9 THE THEORY fef032

9.1 Parents

fef026

9.2 Children

fef033 fef034

9.3 Constants

DenoteConstant
$Class \times ValuedItem\ OPT \to VALUE\_COMP$

MonOp $(ValuedItem\ OPT \to ValuedItem\ OPT)$
$\to VALUE\_COMP$
$\to VALUE\_COMP$

ItemBool $ValuedItem\ OPT \to Bool$

BoolItem $Bool \to ValuedItem\ OPT$

ListAnd $Bool\ LIST \to Bool$

ListOr $Bool\ LIST \to Bool$

ComputeAnd $Class$
$\to (Class \times ValuedItem\ OPT)\ LIST$
$\to Class \times ValuedItem\ OPT$

BinOpAnd $Class \to VALUE\_COMP\ LIST \to VALUE\_COMP$

ComputeOr $Class$
$\to (Class \times ValuedItem\ OPT)\ LIST$
$\to Class \times ValuedItem\ OPT$

BinOpOr $Class \to VALUE\_COMP\ LIST \to VALUE\_COMP$

BinOp $(ValuedItem\ OPT \to ValuedItem\ OPT \to ValuedItem\ OPT)$
$\to VALUE\_COMP$
$\to VALUE\_COMP$
$\to VALUE\_COMP$

TriOp $(ValuedItem\ OPT$
$\to ValuedItem\ OPT$
$\to ValuedItem\ OPT$
$\to ValuedItem\ OPT)$
$\to VALUE\_COMP$
$\to VALUE\_COMP$
$\to VALUE\_COMP$
$\to VALUE\_COMP$

CheckList $Class$
$\to ValuedItem\ OPT$
$\to ((Class \times ValuedItem\ OPT) \times Class \times ValuedItem\ OPT)$
$LIST$
$\to Class$
$\to Class$

CheckTest $Class$
$\rightarrow Class \times ValuedItem\ OPT$
$\rightarrow ((Class \times ValuedItem\ OPT) \times Class \times ValuedItem\ OPT)$
$LIST$
$\rightarrow Class$
$\rightarrow Class$

CaseValValue $ValuedItem\ OPT$
$\rightarrow ((Class \times ValuedItem\ OPT) \times Class \times ValuedItem\ OPT)$
$LIST$
$\rightarrow ValuedItem\ OPT$
$\rightarrow ValuedItem\ OPT$

CaseVal $Class$
$\rightarrow VALUE\_COMP$
$\rightarrow (VALUE\_COMP \times VALUE\_COMP)\ LIST$
$\rightarrow VALUE\_COMP$
$\rightarrow VALUE\_COMP$

CaseC $Class$
$\rightarrow ((Class \times ValuedItem\ OPT) \times Class \times ValuedItem\ OPT)$
$LIST$
$\rightarrow Class$
$\rightarrow Class$

CaseValue $((Class \times ValuedItem\ OPT) \times Class \times ValuedItem\ OPT)$
$LIST$
$\rightarrow ValuedItem\ OPT$
$\rightarrow ValuedItem\ OPT$

Case $Class$
$\rightarrow (VALUE\_COMP \times VALUE\_COMP)\ LIST$
$\rightarrow VALUE\_COMP$
$\rightarrow VALUE\_COMP$

SetFuncAllAnd
$Class \rightarrow VALUE\_COMP \rightarrow VALUE\_COMP$

SetFuncAllOr $Class \rightarrow VALUE\_COMP \rightarrow VALUE\_COMP$

SetFuncAll $(ValuedItem\ OPT\ LIST \rightarrow ValuedItem\ OPT)$
$\rightarrow VALUE\_COMP$
$\rightarrow VALUE\_COMP$

SetFuncDistinct
$(ValuedItem\ OPT\ \mathbb{P} \rightarrow ValuedItem\ OPT)$
$\rightarrow VALUE\_COMP$
$\rightarrow VALUE\_COMP$

NatItem $Worth \rightarrow ValuedItem\ OPT$

CountNonNull $VALUE\_COMP \rightarrow VALUE\_COMP$

CountDistinct
$VALUE\_COMP \rightarrow VALUE\_COMP$

CountAll $VALUE\_COMP$

ExistsTuples $Class \rightarrow TABLE\_COMP \rightarrow VALUE\_COMP$

SingleValue $Class \rightarrow TABLE\_COMP \rightarrow VALUE\_COMP$

Contents $Worth \rightarrow VALUE\_COMP$

ClassItem $Class \rightarrow ValuedItem\ OPT$

Classification	
	$Worth \to VALUE\_COMP$
JoinedRowExistence	
	$Class \to VALUE\_COMP$
JoinSpecs	$DerTableSpec\ LIST \to DerTableSpec$
JoinRows	$DerTableRow \to DerTableRow\ LIST \to DerTableRow\ LIST$
JoinData	$DerTableRow\ LIST\ LIST \to DerTableRow\ LIST$
Join	$DerTable\ LIST \to DerTableSpec \times DerTableRow\ LIST$
ProjectSpec	$DerColSpec\ LIST \to DerTableSpec \to DerTableSpec$
ProjectData	$DerTable\ LIST$ $\to VALUE\_COMP\ LIST$ $\to DerTableRow\ LIST\ LIST$ $\to DerTableRow\ LIST$
Project	$DerTableSpec$ $\to DerTable\ LIST$ $\to (VALUE\_COMP \times DerColSpec)\ LIST$ $\to DerTableRow\ LIST\ LIST$ $\to DerTable$
EvalProjectData	
	$DerTable\ LIST$ $\to VALUE\_COMP\ LIST$ $\to DerTableRow\ LIST\ LIST$ $\to DerTableRow\ LIST$
EvalProject	$DerTableSpec$ $\to DerTable\ LIST$ $\to (VALUE\_COMP \times DerColSpec)\ LIST$ $\to DerTableRow\ LIST\ LIST$ $\to DerTable$
Where	$Class$ $\to DerTable\ LIST$ $\to DerTableRow\ LIST$ $\to VALUE\_COMP$ $\to DerTableRow\ LIST$
PutInGroup	$('a \to 'b) \to 'a \to 'a\ LIST\ LIST \to 'a\ LIST\ LIST$
MakeGroups	$('a \to 'b) \to 'a\ LIST \to 'a\ LIST\ LIST$
ListNth	$Errors \to 'a\ LIST \to 'a\ LIST$
CommonValue	$VALUE\_COMP \to VALUE\_COMP$
Group	$Class$ $\to DerTable\ LIST$ $\to DerTableRow\ LIST$ $\to Errors$ $\to Errors$ $\to VALUE\_COMP$ $\to Class \times DerTableRow\ LIST\ LIST$
TableContents	
	$Worth \to TABLE\_COMP$
AllTuples	$Class$ $\to (VALUE\_COMP \times DerColSpec)\ LIST$

$\rightarrow$ *TABLE_COMP LIST*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *Errors*
$\rightarrow$ *Errors*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *TABLE_COMP*

RemoveDuplicates
$'a\ LIST \rightarrow 'a\ LIST$

DistinctTuples
Class
$\rightarrow$ (*VALUE_COMP* $\times$ *DerColSpec*) *LIST*
$\rightarrow$ *TABLE_COMP LIST*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *Errors*
$\rightarrow$ *Errors*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *TABLE_COMP*

Evaluate *Class*
$\rightarrow$ (*VALUE_COMP* $\times$ *DerColSpec*) *LIST*
$\rightarrow$ *TABLE_COMP LIST*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *Errors*
$\rightarrow$ *Errors*
$\rightarrow$ *VALUE_COMP*
$\rightarrow$ *TABLE_COMP*

$\bigcap_\mathbf{2}$ $'a\ \mathbb{P} \leftrightarrow 'b\ \mathbb{P} \rightarrow 'a\ \mathbb{P} \times 'b\ \mathbb{P}$

ValueComputations
Class $\rightarrow$ *VALUE_COMP* $\mathbb{P}$

TableComputations
Class $\rightarrow$ *TABLE_COMP* $\mathbb{P}$

OkTableComputation
Class $\rightarrow$ *TABLE_COMP* $\mathbb{P}$

OkSTP (*Query* $\rightarrow$ *DerTable LIST* $\rightarrow$ *DerTable* $\times$ *Errors*)
$\rightarrow$ (*Query*, $'$*PARS*) *STP_TYPE* $\mathbb{P}$

OK_TC$_\mathbf{d}$ *Class* $\rightarrow$ *TABLE_COMP* $\mathbb{P}$
OK_VC$_\mathbf{d}$ *Class* $\rightarrow$ *VALUE_COMP* $\mathbb{P}$
OK_VC$_\mathbf{c}$ *Class* $\rightarrow$ *VALUE_COMP* $\mathbb{P}$
OK_TC$_\mathbf{c}$ *Class* $\rightarrow$ *TABLE_COMP* $\mathbb{P}$

9.4 Type Abbreviations

TABLE_COMP *TABLE_COMP*
VALUE_COMP *VALUE_COMP*

9.5 Definitions

DenoteConstant
$\vdash \forall\ ci\bullet\ DenoteConstant\ ci = (\lambda\ tl\ rl\ r\bullet\ ci)$

MonOp $\vdash \forall f\ e$
$\bullet\ MonOp\ f\ e$
$= (\lambda\ tl\ rl\ r$
$\bullet\ (let\ (c,\ v) = e\ tl\ rl\ r\ in\ (c,\ f\ v)))$

ItemBool $\vdash \forall v$
$\bullet\ ItemBool\ v$
$\Leftrightarrow v$
$= ValuedItemItem$
$(MkValuedItem\ sterling\ (BoolVal\ true))$

BoolItem $\vdash \forall v$
$\bullet\ BoolItem\ v$
$= ValuedItemItem$
$(MkValuedItem\ sterling\ (BoolVal\ v))$

ListAnd $\vdash \forall b\ bs$
$\bullet\ (ListAnd\ []\ \Leftrightarrow true)$
$\wedge\ (ListAnd\ (Cons\ b\ bs) \Leftrightarrow b \wedge ListAnd\ bs)$

ListOr $\vdash \forall b\ bs$
$\bullet\ (ListOr\ []\ \Leftrightarrow false)$
$\wedge\ (ListOr\ (Cons\ b\ bs) \Leftrightarrow b \vee ListOr\ bs)$

ComputeAnd $\vdash \forall cc\ cil$
$\bullet\ ComputeAnd\ cc\ cil$
$= (let\ hcil$
$= cil$
$\upharpoonright \{(c,\ i) | cc\ dominates\ c \wedge \neg\ ItemBool\ i\}$
$in\ let\ v \Leftrightarrow ListAnd\ (Map\ (ItemBool\ o\ Snd)\ cil)$
$in\ let\ makecase\ (c,\ u)$
$= (if\ ItemBool\ u$
$then\ lattice\_bottom$
$else\ c)$
$in\ if\ hcil = []$
$then\ (lubl\ (Map\ Fst\ cil),\ BoolItem\ v)$
$else$
$(lubl\ (Map\ makecase\ hcil),$
$BoolItem\ false))$

BinOpAnd $\vdash \forall cc\ el$
$\bullet\ BinOpAnd\ cc\ el$
$= (\lambda\ tl\ rl\ r$
$\bullet\ ComputeAnd\ cc\ (Map\ (\lambda\ e\bullet\ e\ tl\ rl\ r)\ el))$

ComputeOr $\vdash \forall cc\ cil$
$\bullet\ ComputeOr\ cc\ cil$
$= (let\ hcil$
$= cil$
$\upharpoonright \{(c,\ i) | cc\ dominates\ c \wedge ItemBool\ i\}$
$in\ let\ v \Leftrightarrow ListOr\ (Map\ (ItemBool\ o\ Snd)\ cil)$
$in\ let\ makecase\ (c,\ u)$
$= (if\ \neg\ ItemBool\ u$
$then\ lattice\_bottom$
$else\ c)$

$$
\begin{array}{ll}
 & \quad\quad\quad\quad \mathit{in\ if\ \ hcil} = [] \\
 & \quad\quad\quad\quad \mathit{then}\ (\mathit{lubl}\ (\mathit{Map\ Fst\ cil}),\ \mathit{BoolItem\ v}) \\
 & \quad\quad\quad\quad \mathit{else} \\
 & \quad\quad\quad\quad\ \ (\mathit{lubl}\ (\mathit{Map\ makecase\ hcil}), \\
 & \quad\quad\quad\quad\quad \mathit{BoolItem\ true})) \\
\textbf{BinOpOr} & \vdash \forall\ \mathit{cc\ el} \\
 & \ \bullet\ \mathit{BinOpOr\ cc\ el} \\
 & \quad\quad = (\lambda\ \mathit{tl\ rl\ r} \\
 & \quad\quad \bullet\ \mathit{ComputeOr\ cc}\ (\mathit{Map}\ (\lambda\ e\bullet\ e\ \mathit{tl\ rl\ r})\ \mathit{el})) \\
\textbf{BinOp} & \vdash \forall\ f\ \mathit{e1\ e2} \\
 & \ \bullet\ \mathit{BinOp}\ f\ \mathit{e1\ e2} \\
 & \quad\quad = (\lambda\ \mathit{tl\ rl\ r} \\
 & \quad\quad \bullet\ (\mathit{let}\ (\mathit{c1},\ \mathit{v1}) = \mathit{e1\ tl\ rl\ r} \\
 & \quad\quad\quad \mathit{in\ let}\ (\mathit{c2},\ \mathit{v2}) = \mathit{e2\ tl\ rl\ r} \\
 & \quad\quad\quad\quad \mathit{in}\ (\mathit{c1\ lub\ c2},\ f\ \mathit{v1\ v2}))) \\
\textbf{TriOp} & \vdash \forall\ f\ \mathit{e1\ e2\ e3} \\
 & \ \bullet\ \mathit{TriOp}\ f\ \mathit{e1\ e2\ e3} \\
 & \quad\quad = (\lambda\ \mathit{tl\ rl\ r} \\
 & \quad\quad \bullet\ (\mathit{let}\ (\mathit{c1},\ \mathit{v1}) = \mathit{e1\ tl\ rl\ r} \\
 & \quad\quad\quad \mathit{in\ let}\ (\mathit{c2},\ \mathit{v2}) = \mathit{e2\ tl\ rl\ r} \\
 & \quad\quad\quad\quad \mathit{in\ let}\ (\mathit{c3},\ \mathit{v3}) = \mathit{e3\ tl\ rl\ r} \\
 & \quad\quad\quad\quad\quad \mathit{in}\ (\mathit{c1\ lub\ c2\ lub\ c3},\ f\ \mathit{v1\ v2\ v3}))) \\
\textbf{CheckList} & \vdash \forall\ \mathit{cc\ ti\ cv\ cvs\ elsec} \\
 & \ \bullet\ \mathit{CheckList\ cc\ ti}\ []\ \mathit{elsec} = \mathit{elsec} \\
 & \quad\quad \wedge\ \mathit{CheckList\ cc\ ti}\ (\mathit{Cons\ cv\ cvs})\ \mathit{elsec} \\
 & \quad\quad\quad = (\mathit{let}\ ((\mathit{cec},\ \mathit{cei}),\ \mathit{vec},\ \mathit{vei}) = \mathit{cv} \\
 & \quad\quad\quad\quad \mathit{in\ if}\ \neg\ \mathit{cc\ dominates\ cec} \\
 & \quad\quad\quad\quad \mathit{then\ cec} \\
 & \quad\quad\quad\quad \mathit{else\ if\ \ ti} = \mathit{cei} \\
 & \quad\quad\quad\quad \mathit{then\ vec} \\
 & \quad\quad\quad\quad \mathit{else\ CheckList\ cc\ ti\ cvs\ elsec}) \\
\textbf{CheckTest} & \vdash \forall\ \mathit{cc\ ti\ tc\ cvs\ elsec} \\
 & \ \bullet\ \mathit{CheckTest\ cc}\ (\mathit{tc},\ \mathit{ti})\ \mathit{cvs\ elsec} \\
 & \quad\quad = (\mathit{if\ \ cc\ dominates\ tc} \\
 & \quad\quad\quad \mathit{then\ CheckList\ cc\ ti\ cvs\ elsec} \\
 & \quad\quad\quad \mathit{else\ tc}) \\
\textbf{CaseValValue} & \vdash \forall\ \mathit{ti\ cv\ cvs\ elsev} \\
 & \ \bullet\ \mathit{CaseValValue\ ti}\ []\ \mathit{elsev} = \mathit{elsev} \\
 & \quad\quad \wedge\ \mathit{CaseValValue\ ti}\ (\mathit{Cons\ cv\ cvs})\ \mathit{elsev} \\
 & \quad\quad\quad = (\mathit{let}\ ((\mathit{cec},\ \mathit{cei}),\ \mathit{vec},\ \mathit{vei}) = \mathit{cv} \\
 & \quad\quad\quad\quad \mathit{in\ if\ \ ti} = \mathit{cei} \\
 & \quad\quad\quad\quad \mathit{then\ vei} \\
 & \quad\quad\quad\quad \mathit{else\ CaseValValue\ ti\ cvs\ elsev}) \\
\textbf{CaseVal} & \vdash \forall\ \mathit{cc\ tst\ casevals\ elseval} \\
 & \ \bullet\ \mathit{CaseVal\ cc\ tst\ casevals\ elseval} \\
 & \quad\quad = (\lambda\ \mathit{tl\ rl\ r} \\
 & \quad\quad \bullet\ (\mathit{let}\ (\mathit{tc},\ \mathit{ti}) = \mathit{tst\ tl\ rl\ r} \\
 & \quad\quad\quad \mathit{in\ let\ cvs}
\end{array}
$$

$= Map$
$(\lambda\ (c,\ v) \bullet\ (c\ tl\ rl\ r,\ v\ tl\ rl\ r))$
$casevals$
$in\ let\ (ec,\ ei) = elseval\ tl\ rl\ r$
$in\ let\ c = CheckTest\ cc\ (tc,\ ti)\ cvs\ ec$
$in\ let\ v = CaseValValue\ ti\ cvs\ ei$
$in\ (c,\ v)))$

CaseC $\vdash \forall\ cc\ cv\ cvs\ elsec$
$\bullet\ CaseC\ cc\ []\ elsec = elsec$
$\wedge\ CaseC\ cc\ (Cons\ cv\ cvs)\ elsec$
$= (let\ ((cec,\ cei),\ vec,\ vei) = cv$
$in\ if\ \neg\ cc\ dominates\ cec$
$then\ cec$
$else\ if\ ItemBool\ cei$
$then\ vec$
$else\ CaseC\ cc\ cvs\ elsec)$

CaseValue $\vdash \forall\ cv\ cvs\ elsev$
$\bullet\ CaseValue\ []\ elsev = elsev$
$\wedge\ CaseValue\ (Cons\ cv\ cvs)\ elsev$
$= (let\ ((cec,\ cei),\ vec,\ vei) = cv$
$in\ if\ ItemBool\ cei$
$then\ vei$
$else\ CaseValue\ cvs\ elsev)$

Case $\vdash \forall\ cc\ casevals\ elseval$
$\bullet\ Case\ cc\ casevals\ elseval$
$= (\lambda\ tl\ rl\ r$
$\bullet\ (let\ cvs$
$= Map$
$(\lambda\ (c,\ v) \bullet\ (c\ tl\ rl\ r,\ v\ tl\ rl\ r))$
$casevals$
$in\ let\ (ec,\ ei) = elseval\ tl\ rl\ r$
$in\ let\ c = CaseC\ cc\ cvs\ ec$
$in\ let\ v = CaseValue\ cvs\ ei\ in\ (c,\ v)))$

SetFuncAllAnd
$\vdash \forall\ cc\ e$
$\bullet\ SetFuncAllAnd\ cc\ e$
$= (\lambda\ tl\ rl\ r \bullet\ ComputeAnd\ cc\ (Map\ (e\ tl\ rl)\ rl))$

SetFuncAllOr $\vdash \forall\ cc\ e$
$\bullet\ SetFuncAllOr\ cc\ e$
$= (\lambda\ tl\ rl\ r \bullet\ ComputeOr\ cc\ (Map\ (e\ tl\ rl)\ rl))$

SetFuncAll $\vdash \forall\ f\ e$
$\bullet\ SetFuncAll\ f\ e$
$= (\lambda\ tl\ rl\ r$
$\bullet\ (let\ (cl,\ il) = Split\ (Map\ (e\ tl\ rl)\ rl)$
$in\ (lubl\ cl,\ f\ il)))$

SetFuncDistinct
$\vdash \forall\ f\ e$
$\bullet\ SetFuncDistinct\ f\ e$

$$
\begin{array}{ll}
 & \quad = (\lambda\ tl\ rl\ r \\
 & \quad \bullet (let\ (cl,\ il) = Split\ (Map\ (e\ tl\ rl)\ rl) \\
 & \qquad in\ (lubl\ cl,\ f\ (Elems\ il)))) \\
\textbf{NatItem} & \vdash true \\
\textbf{CountNonNull} & \vdash \forall\ e \\
 & \bullet\ CountNonNull\ e \\
 & \quad = (let\ counter\ il \\
 & \qquad = NatItem\ (\#\ (il \upharpoonright \{i | isValuedItem\ i\})) \\
 & \quad in\ SetFuncAll\ counter\ e) \\
\textbf{CountDistinct} & \\
 & \vdash \forall\ e \\
 & \bullet\ CountDistinct\ e \\
 & \quad = (let\ counter\ is \\
 & \qquad = NatItem\ (\#\ (is \cap \{i | isValuedItem\ i\})) \\
 & \quad in\ SetFuncDistinct\ counter\ e) \\
\textbf{CountAll} & \vdash CountAll \\
 & \quad = (\lambda\ tl\ rl\ r \\
 & \quad \bullet (let\ cl = Map\ DTR\_row\ rl \\
 & \qquad in\ (lubl\ cl,\ NatItem\ (\#\ rl)))) \\
\textbf{ExistsTuples} & \vdash \forall\ cc\ te \\
 & \bullet\ ExistsTuples\ cc\ te \\
 & \quad = (\lambda\ tl\ rl\ r \\
 & \quad \bullet (let\ (c,\ t) = te\ tl \\
 & \qquad in\ let\ trl \\
 & \qquad\qquad = DT\_rows\ t \\
 & \qquad\qquad\ \upharpoonright \{r \\
 & \qquad\qquad\ | cc\ dominates\ DTR\_row\ r \\
 & \qquad\qquad\quad \wedge cc\ dominates\ DTR\_where\ r\} \\
 & \qquad in\ if\ cc\ dominates\ c \\
 & \qquad then \\
 & \qquad\ (lubl\ (Map\ DTR\_row\ trl), \\
 & \qquad\quad BoolItem\ (\neg\ trl = [])) \\
 & \qquad else\ (c,\ Arbitrary))) \\
\textbf{SingleValue} & \vdash \forall\ cc\ te \\
 & \bullet\ SingleValue\ cc\ te \\
 & \quad = (\lambda\ tl\ rl\ r \\
 & \quad \bullet (let\ (c,\ t) = te\ tl \\
 & \qquad in\ let\ trl \\
 & \qquad\qquad = DT\_rows\ t \\
 & \qquad\qquad\ \upharpoonright \{r \\
 & \qquad\qquad\ | cc\ dominates\ DTR\_row\ r \\
 & \qquad\qquad\quad \wedge cc\ dominates\ DTR\_where\ r\} \\
 & \qquad in\ let\ cil = DTR\_cols\ (Head\ trl) \\
 & \qquad\ in\ let\ (ic,\ ii) = Head\ cil \\
 & \qquad\quad in\ if\ cc\ dominates\ c \\
 & \qquad\quad then \\
 & \qquad\qquad if\ \#\ trl = 1 \wedge \#\ cil = 1 \\
 & \qquad\qquad then\ (ic,\ ii)
\end{array}
$$

```
                    else (c, Arbitrary)
                  else (c, Arbitrary)))
```

Contents

$\vdash \forall\ i$
$\bullet\ Contents\ i$
$= (\lambda\ tl\ rl\ r$
$\bullet\ if\ 1 \leq i \land i \leq \#\ (DTR\_cols\ r)$
$then\ Nth\ (DTR\_cols\ r)\ i$
$else\ Arbitrary)$

ClassItem

$\vdash \forall\ v$
$\bullet\ ClassItem\ v$
$=\ ValuedItemItem$
$(MkValuedItem\ sterling\ (ClassVal\ v))$

Classification

$\vdash \forall\ i$
$\bullet\ Classification\ i$
$= (\lambda\ tl\ rl\ r$
$\bullet\ if\ 1 \leq i \land i \leq \#\ (DTR\_cols\ r)$
$then$
$(DTR\_row\ r,$
$ClassItem\ (Fst\ (Nth\ (DTR\_cols\ r)\ i)))$
$else\ Arbitrary)$

JoinedRowExistence

$\vdash \forall\ cc$
$\bullet\ JoinedRowExistence\ cc$
$= (\lambda\ tl\ rl\ r \bullet\ (cc,\ ClassItem\ (DTR\_row\ r)))$

JoinSpecs

$\vdash \forall\ sl$
$\bullet\ JoinSpecs\ sl$
$= (let\ n = []$
$and\ mr = lubl\ (Map\ DTS\_maxRow\ sl)$
$and\ csl = Flat\ (Map\ DTS\_colSpecs\ sl)$
$in\ MkDerTableSpec\ n\ mr\ csl)$

JoinRows

$\vdash \forall\ r\ rs$
$\bullet\ JoinRows\ r\ rs$
$= (let\ join2\ rr$
$= MkDerTableRow$
$(DTR\_where\ r\ lub\ DTR\_where\ rr)$
$(DTR\_row\ r\ lub\ DTR\_row\ rr)$
$(DTR\_cols\ r\ @\ DTR\_cols\ rr)$
$in\ Map\ join2\ rs)$

JoinData

$\vdash JoinData\ [] = []$
$\land (\forall\ tab\ rest$
$\bullet\ JoinData\ (Cons\ tab\ rest)$
$= (if\ rest = []$
$then\ tab$
$else$
$(let\ jrest = JoinData\ rest$
$in\ let\ join\_blk\ r = JoinRows\ r\ jrest$
$in\ Flat\ (Map\ join\_blk\ tab))))$

Join $\vdash \forall\ tabl$
$\bullet\ Join\ tabl$
$= (JoinSpecs\ (Map\ DT\_spec\ tabl),$
$JoinData\ (Map\ DT\_rows\ tabl))$

ProjectSpec $\vdash \forall\ sl\ s$
$\bullet\ ProjectSpec\ sl\ s$
$= MkDerTableSpec\ []\ (DTS\_maxRow\ s)\ sl$

ProjectData $\vdash \forall\ tl\ el\ gps$
$\bullet\ ProjectData\ tl\ el\ gps$
$= (let\ h\ gp\ r$
$= MkDerTableRow$
$(DTR\_where\ r)$
$(DTR\_row\ r)$
$(Map\ (\lambda\ e \bullet\ e\ tl\ gp\ r)\ el)$
$in\ let\ k\ gp = Map\ (h\ gp)\ gp$
$in\ Flat\ (Map\ k\ gps))$

Project $\vdash \forall\ ts\ tl\ sellist\ gps$
$\bullet\ Project\ ts\ tl\ sellist\ gps$
$= MkDerTable$
$(ProjectSpec\ (Map\ Snd\ sellist)\ ts)$
$(ProjectData\ tl\ (Map\ Fst\ sellist)\ gps)$

EvalProjectData
$\vdash \forall\ tl\ el\ gps$
$\bullet\ EvalProjectData\ tl\ el\ gps$
$= (let\ h\ gp\ r$
$= MkDerTableRow$
$(DTR\_where\ r)$
$(DTR\_row\ r)$
$(Map\ (\lambda\ e \bullet\ e\ tl\ gp\ r)\ el)$
$in\ let\ k\ gp$
$= (let\ results = Map\ (h\ gp)\ gp$
$in\ if\ \#\ (Elems\ results) = 1$
$then\ Head\ results$
$else\ Arbitrary)\ in\ Map\ k\ gps)$

EvalProject $\vdash \forall\ ts\ tl\ sellist\ gps$
$\bullet\ EvalProject\ ts\ tl\ sellist\ gps$
$= MkDerTable$
$(ProjectSpec\ (Map\ Snd\ sellist)\ ts)$
$(EvalProjectData\ tl\ (Map\ Fst\ sellist)\ gps)$

Where $\vdash \forall\ c\ tl\ rl\ e$
$\bullet\ Where\ c\ tl\ rl\ e$
$= (let\ hrl = rl \upharpoonright \{r | c\ dominates\ DTR\_row\ r\}$
$in\ let\ w\ r = DTR\_where\ r\ lub\ Fst\ (e\ tl\ hrl\ r)$
$in\ let\ h\ r$
$= ((ItemBool\ (Snd\ (e\ tl\ hrl\ r))$
$\lor \neg\ c\ dominates\ w\ r),$
$MkDerTableRow$
$(w\ r)$

$$
\begin{array}{ll}
 & \quad\quad\quad\quad (DTR\_row\ r) \\
 & \quad\quad\quad\quad (DTR\_cols\ r)) \\
 & \quad\quad\quad in\ Map\ Snd\ (Map\ h\ hrl \restriction \{(t,\ r)|t\})) \\
\textbf{PutInGroup} & \vdash \forall\ gpby\ x\ gp\ gps \\
 & \bullet\ PutInGroup\ gpby\ x\ []\ =\ [[x]] \\
 & \quad \wedge\ PutInGroup\ gpby\ x\ (Cons\ gp\ gps) \\
 & \quad\quad = (if\ gpby\ x\ =\ gpby\ (Head\ gp) \\
 & \quad\quad\quad then\ Cons\ (Cons\ x\ gp)\ gps \\
 & \quad\quad\quad else\ Cons\ gp\ (PutInGroup\ gpby\ x\ gps)) \\
\textbf{MakeGroups} & \vdash \forall\ gpby\ x\ xs \\
 & \bullet\ MakeGroups\ gpby\ []\ =\ [] \\
 & \quad \wedge\ MakeGroups\ gpby\ (Cons\ x\ xs) \\
 & \quad\quad = PutInGroup\ gpby\ x\ (MakeGroups\ gpby\ xs) \\
\textbf{ListNth} & \vdash \forall\ n\ nl\ list \\
 & \bullet\ ListNth\ []\ list\ =\ [] \\
 & \quad \wedge\ ListNth\ (Cons\ n\ nl)\ list \\
 & \quad\quad = Cons \\
 & \quad\quad\quad (if\ 1 \le n \wedge n \le \#\ list \\
 & \quad\quad\quad\quad then\ Nth\ list\ n \\
 & \quad\quad\quad\quad else\ Arbitrary) \\
 & \quad\quad\quad (ListNth\ nl\ list) \\
\textbf{CommonValue} & \vdash \forall\ e \\
 & \bullet\ CommonValue\ e \\
 & \quad = (let\ pick\ il \\
 & \quad\quad\quad = (if\ \#\ (Elems\ il)\ =\ 1 \\
 & \quad\quad\quad\quad then\ Head\ il \\
 & \quad\quad\quad\quad else\ Arbitrary)\ in\ SetFuncAll\ pick\ e) \\
\textbf{Group} & \vdash \forall\ cc\ tl\ rl\ gbsterling\ gbclass\ having \\
 & \bullet\ Group\ cc\ tl\ rl\ gbsterling\ gbclass\ having \\
 & \quad = (let\ gpby\ row \\
 & \quad\quad\quad = (ListNth \\
 & \quad\quad\quad\quad\quad gbsterling \\
 & \quad\quad\quad\quad\quad (Map\ Snd\ (DTR\_cols\ row)), \\
 & \quad\quad\quad\quad ListNth\ gbclass\ (Map\ Fst\ (DTR\_cols\ row))) \\
 & \quad\quad in\ let\ gbc\ row \\
 & \quad\quad\quad\quad = lubl \\
 & \quad\quad\quad\quad\quad (ListNth \\
 & \quad\quad\quad\quad\quad\quad gbsterling \\
 & \quad\quad\quad\quad\quad\quad (Map\ Fst\ (DTR\_cols\ row))) \\
 & \quad\quad\quad in\ let\ gps\ =\ MakeGroups\ gpby\ rl \\
 & \quad\quad\quad\quad in\ let\ has\_test\ gp \\
 & \quad\quad\quad\quad\quad\quad = CommonValue\ having\ tl\ gp\ Arbitrary \\
 & \quad\quad\quad\quad\quad in\ let\ cl \\
 & \quad\quad\quad\quad\quad\quad\quad = (if \\
 & \quad\quad\quad\quad\quad\quad\quad\quad cc\ dominates\ lubl\ (Map\ gbc\ rl) \\
 & \quad\quad\quad\quad\quad\quad\quad then \\
 & \quad\quad\quad\quad\quad\quad\quad\quad lubl\ (Map\ (Fst\ o\ has\_test)\ gps) \\
 & \quad\quad\quad\quad\quad\quad\quad else\ lubl\ (Map\ gbc\ rl))
\end{array}
$$

$$
\begin{array}{l}
\quad in\ let\ wanted\_gps \\
\qquad = gps \\
\qquad\quad \upharpoonright \{gp \\
\qquad\quad | ItemBool\ (Snd\ (has\_test\ gp))\} \\
\quad in\ (cl,\ wanted\_gps))
\end{array}
$$

TableContents

$$
\begin{array}{l}
\vdash \forall\ i \\
\quad \bullet\ TableContents\ i \\
\qquad = (\lambda\ tl \\
\qquad \bullet\ if\ 1\ \leq\ i\ \wedge\ i\ \leq\ \#\ tl \\
\qquad\quad then\ (lattice\_bottom,\ Nth\ tl\ i) \\
\qquad\quad else\ Arbitrary)
\end{array}
$$

AllTuples

$$
\begin{array}{l}
\vdash \forall\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having \\
\quad \bullet\ AllTuples \\
\qquad\quad cc \\
\qquad\quad sellist \\
\qquad\quad fromspec \\
\qquad\quad where \\
\qquad\quad gbsterling \\
\qquad\quad gbclass \\
\qquad\quad having \\
\qquad = (\lambda\ tl \\
\qquad \bullet\ (let\ (cll,\ tabs) \\
\qquad\qquad\quad = Split\ (Map\ (\lambda\ te \bullet\ te\ tl)\ fromspec) \\
\qquad\quad in\ let\ (ts,\ tab1)\ =\ Join\ tabs \\
\qquad\quad\ in\ let\ tab2\ =\ Where\ cc\ tl\ tab1\ where \\
\qquad\qquad in\ let\ (cl1,\ gps) \\
\qquad\qquad\qquad = Group \\
\qquad\qquad\qquad\ \ cc \\
\qquad\qquad\qquad\ \ tl \\
\qquad\qquad\qquad\ \ tab2 \\
\qquad\qquad\qquad\ \ gbsterling \\
\qquad\qquad\qquad\ \ gbclass \\
\qquad\qquad\qquad\ \ having \\
\qquad\qquad\ in\ let\ cl2 \\
\qquad\qquad\qquad = (if\ cc\ dominates\ lubl\ cll \\
\qquad\qquad\qquad\quad then\ cl1 \\
\qquad\qquad\qquad\quad else\ lubl\ cll) \\
\qquad\qquad\quad in\ (cl2,\ Project\ ts\ tl\ sellist\ gps)))
\end{array}
$$

RemoveDuplicates

$$
\begin{array}{l}
\vdash \forall\ x\ xs \\
\quad \bullet\ RemoveDuplicates\ []\ =\ [] \\
\qquad \wedge\ RemoveDuplicates\ (Cons\ x\ xs) \\
\qquad\quad = Cons\ x\ (RemoveDuplicates\ xs\ \upharpoonright\ \{y | \neg\ y\ =\ x\})
\end{array}
$$

DistinctTuples

$$
\begin{array}{l}
\vdash \forall\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having \\
\quad \bullet\ DistinctTuples \\
\qquad\quad cc
\end{array}
$$

sellist
fromspec
where
gbsterling
gbclass
having
$= (\lambda\ tl$
$\bullet\ (let\ (cll,\ tabs)$
$= Split\ (Map\ (\lambda\ te \bullet\ te\ tl)\ fromspec)$
$in\ let\ (ts,\ tab1) = Join\ tabs$
$in\ let\ tab2 = Where\ cc\ tl\ tab1\ where$
$in\ let\ (cl,\ gps)$
$= Group$
cc
tl
tab2
gbsterling
gbclass
having
$in\ let\ rem\_dups\ tab$
$= MkDerTable$
$(DT\_spec\ tab)$
$(RemoveDuplicates$
$(DT\_rows\ tab))$
$in\ (cl\ lub\ lubl\ cll,$
rem_dups
$(Project\ ts\ tl\ sellist\ gps))))$

Evaluate $\vdash \forall\ cc\ sellist\ fromspec\ where\ gbsterling\ gbclass\ having$
$\bullet\ Evaluate$
cc
sellist
fromspec
where
gbsterling
gbclass
having
$= (\lambda\ tl$
$\bullet\ (let\ (cll,\ tabs)$
$= Split\ (Map\ (\lambda\ te \bullet\ te\ tl)\ fromspec)$
$in\ let\ (ts,\ tab1) = Join\ tabs$
$in\ let\ tab2 = Where\ cc\ tl\ tab1\ where$
$in\ let\ (cl,\ gps)$
$= Group$
cc
tl
tab2
gbsterling
gbclass

$$
\begin{array}{l}
\quad\quad\quad\quad \mathit{having} \\
\quad\quad\quad \mathit{in}\ (\mathit{cl\ lub\ lubl\ cll}, \\
\quad\quad\quad\quad \mathit{EvalProject\ ts\ tl\ sellist\ gps})))
\end{array}
$$

$\bigcap_{\mathbf{2}}$

$$
\begin{array}{l}
\vdash \forall\ u \\
\quad \bullet \bigcap_2\ u \\
\quad\quad = (\bigcap\ \{a | \exists\ b \bullet\ (a,\ b) \in u\},\ \bigcap\ \{b | \exists\ a \bullet\ (a,\ b) \in u\})
\end{array}
$$

TableComputations
ValueComputations

$$
\begin{array}{l}
\vdash \mathit{ConstSpec} \\
\quad (\lambda\ (\mathit{TableComputations}',\ \mathit{ValueComputations}') \\
\quad\quad \bullet \forall\ cc \\
\quad\quad\quad \bullet (\mathit{TableComputations}'\ cc, \\
\quad\quad\quad\quad\quad \mathit{ValueComputations}'\ cc) \\
\quad\quad\quad\quad = \bigcap_2 \\
\quad\quad\quad\quad\quad \{(tes,\ es) \\
\quad\quad\quad\quad\quad\quad | (\forall\ ci \bullet\ \mathit{DenoteConstant}\ ci \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ i \bullet\ \mathit{Contents}\ i \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ i \bullet\ \mathit{Classification}\ i \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathit{CountAll} \in es \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ f\ e \bullet\ e \in es \Rightarrow \mathit{MonOp}\ f\ e \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ f\ e1\ e2 \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ e1 \in es \wedge e2 \in es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{BinOp}\ f\ e1\ e2 \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ f\ e1\ e2\ e3 \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ e1 \in es \wedge e2 \in es \wedge e3 \in es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{TriOp}\ f\ e1\ e2\ e3 \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ el \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ \mathit{Elems}\ el \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{BinOpAnd}\ cc\ el \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ el \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ \mathit{Elems}\ el \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{BinOpOr}\ cc\ el \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ te\ cel\ ee \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ te \in es \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathit{Elems}\ (\mathit{Map}\ \mathit{Fst}\ cel) \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathit{Elems}\ (\mathit{Map}\ \mathit{Snd}\ cel) \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ ee \in es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{CaseVal}\ cc\ te\ cel\ ee \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ cel\ ee \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ \mathit{Elems}\ (\mathit{Map}\ \mathit{Fst}\ cel) \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ \mathit{Elems}\ (\mathit{Map}\ \mathit{Snd}\ cel) \subseteq es \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\ ee \in es \\
\quad\quad\quad\quad\quad\quad\quad\quad \Rightarrow \mathit{Case}\ cc\ cel\ ee \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ e \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ e \in es \Rightarrow \mathit{SetFuncAllAnd}\ cc\ e \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ e \\
\quad\quad\quad\quad\quad\quad\quad \bullet\ e \in es \Rightarrow \mathit{SetFuncAllOr}\ cc\ e \in es) \\
\quad\quad\quad\quad\quad\quad\quad \wedge (\forall\ e
\end{array}
$$

$$
\begin{array}{l}
\bullet\ e \in es \Rightarrow CountNonNull\ e \in es) \\
\wedge\ (\forall\ e \\
\bullet\ e \in es \Rightarrow CountDistinct\ e \in es) \\
\wedge\ (\forall\ e \\
\bullet\ e \in es \Rightarrow CommonValue\ e \in es) \\
\wedge\ (\forall\ f\ e \\
\bullet\ e \in es \Rightarrow SetFuncAll\ f\ e \in es) \\
\wedge\ (\forall\ f\ e \\
\bullet\ e \in es \\
\quad \Rightarrow SetFuncDistinct\ f\ e \in es) \\
\wedge\ (\forall\ te \\
\bullet\ te \in tes \\
\quad \Rightarrow ExistsTuples\ cc\ te \in es) \\
\wedge\ (\forall\ te \\
\bullet\ te \in tes \\
\quad \Rightarrow SingleValue\ cc\ te \in es) \\
\wedge\ JoinedRowExistence\ cc \in es \\
\wedge\ (\forall\ i \bullet\ TableContents\ i \in tes) \\
\wedge\ (\forall\ esl\ tel\ e1\ ml\ nl\ e2 \\
\bullet\ Elems\ (Map\ Fst\ esl) \subseteq es \\
\qquad \wedge\ Elems\ tel \subseteq tes \\
\qquad \wedge\ e1 \in es \\
\qquad \wedge\ e2 \in es \\
\quad \Rightarrow AllTuples \\
\qquad cc \\
\qquad esl \\
\qquad tel \\
\qquad e1 \\
\qquad ml \\
\qquad nl \\
\qquad e2 \\
\qquad \in tes)\})
\end{array}
$$

$(TableComputations,\ ValueComputations)$

OkTableComputation

$$
\begin{array}{l}
\vdash \forall\ cc\ te \\
\quad \bullet\ te \in OkTableComputation\ cc \\
\qquad \Leftrightarrow (let\ c\ tl = Fst\ (te\ tl) \\
\qquad\quad in\ let\ f\ tl = (Snd\ (te\ tl),\ []) \\
\qquad\qquad in\ RiskInputs\ cc\ f \\
\qquad\qquad\quad \subseteq \{tl | \neg\ cc\ dominates\ c\ tl\})
\end{array}
$$

OkSTP

$$
\begin{array}{l}
\vdash \forall\ compile\ stp \\
\quad \bullet\ stp \in OkSTP\ compile \\
\qquad \Leftrightarrow (\forall\ q\ c \\
\qquad \bullet\ isError\ (stp\ (q,\ c)) \\
\qquad\quad \vee\ (let\ (dq,\ ocq,\ pars) = destVal\ (stp\ (q,\ c)) \\
\qquad\qquad in\ \exists\ dte \\
\qquad\qquad \bullet\ dte \in TableComputations\ c \\
\qquad\qquad\quad \wedge\ compile\ dq
\end{array}
$$

$$
\begin{array}{l}
\quad = (\lambda\ tl \bullet\ (Snd\ (dte\ tl),\ []))\\
\quad \land\ (\forall\ tl\\
\quad \bullet\ \neg\ c\ dominates\ Fst\ (dte\ tl)\\
\qquad \Rightarrow IsL\ ocq\\
\qquad\quad \land\ is\_select\ (OutL\ ocq)\\
\qquad\quad \land\ \neg\ DT\_rows\\
\qquad\qquad (Fst\\
\qquad\qquad\quad (compile\\
\qquad\qquad\qquad (OutL\ ocq)\\
\qquad\qquad\qquad tl))\\
\qquad\qquad = [])))
\end{array}
$$

OK_TC$_\mathbf{d}$

$$
\begin{array}{l}
\vdash \forall\ c\ tc\\
\bullet\ tc \in OK\_TC_d\ c\\
\quad \Leftrightarrow (\forall\ tl_0\ tl_1\\
\quad \bullet\ Map\ (HideDerTable\ c)\ tl_0\\
\qquad\quad = Map\ (HideDerTable\ c)\ tl_1\\
\qquad \land\ \neg\ HideDerTable\ c\ (Snd\ (tc\ tl_0))\\
\qquad\quad = HideDerTable\ c\ (Snd\ (tc\ tl_1))\\
\qquad \Rightarrow \neg\ c\ dominates\ Fst\ (tc\ tl_0))
\end{array}
$$

OK_VC$_\mathbf{d}$

$$
\begin{array}{l}
\vdash \forall\ c\ vc\\
\bullet\ vc \in OK\_VC_d\ c\\
\quad \Leftrightarrow (\forall\ tl_0\ tl_1\ rl_0\ rl_1\ r_0\ r_1\\
\quad \bullet\ Map\ (HideDerTable\ c)\ tl_0\\
\qquad\quad = Map\ (HideDerTable\ c)\ tl_1\\
\qquad \land\ Map\ (HideDerTableRow\ c)\ rl_0\\
\qquad\quad = Map\ (HideDerTableRow\ c)\ rl_1\\
\qquad \land\ HideDerTableRow\ c\ r_0\\
\qquad\quad = HideDerTableRow\ c\ r_1\\
\qquad \land\ \neg\ Snd\ (vc\ tl_0\ rl_0\ r_0)\\
\qquad\quad = Snd\ (vc\ tl_1\ rl_1\ r_1)\\
\qquad \Rightarrow \neg\ c\ dominates\ Fst\ (vc\ tl_0\ rl_0\ r_0))
\end{array}
$$

OK_VC$_\mathbf{c}$

$$
\begin{array}{l}
\vdash \forall\ c\ vc\\
\bullet\ vc \in OK\_VC_c\ c\\
\quad \Leftrightarrow (\forall\ tl_0\ tl_1\ rl_0\ rl_1\ r_0\ r_1\\
\quad \bullet\ Map\ (HideDerTable\ c)\ tl_0\\
\qquad\quad = Map\ (HideDerTable\ c)\ tl_1\\
\qquad \land\ Map\ (HideDerTableRow\ c)\ rl_0\\
\qquad\quad = Map\ (HideDerTableRow\ c)\ rl_1\\
\qquad \land\ HideDerTableRow\ c\ r_0\\
\qquad\quad = HideDerTableRow\ c\ r_1\\
\qquad \Rightarrow Fst\ (vc\ tl_0\ rl_0\ r_0)\\
\qquad\quad = Fst\ (vc\ tl_1\ rl_1\ r_1))
\end{array}
$$

OK_TC$_\mathbf{c}$

$$
\begin{array}{l}
\vdash \forall\ c\ tc\\
\bullet\ tc \in OK\_TC_c\ c\\
\quad \Leftrightarrow (\forall\ tl_0\ tl_1\\
\quad \bullet\ Map\ (HideDerTable\ c)\ tl_0\\
\qquad\quad = Map\ (HideDerTable\ c)\ tl_1\\
\qquad \Rightarrow Fst\ (tc\ tl_0) = Fst\ (tc\ tl_1))
\end{array}
$$

10 INDEX