Maintaining Database Integrity with Refinement Types

Ioannis G. Baltopoulos[1], Johannes Borgström[2], and Andrew D. Gordon[2]

[1] University of Cambridge
[2] Microsoft Research

Abstract. Given recent advances in automated theorem proving, we present a new method for determining whether database transactions preserve integrity constraints. We consider check constraints and referential-integrity constraints—extracted from SQL table declarations—and application-level invariants expressed as formulas of first-order logic. Our motivation is to use static analysis of database transactions at development time, to catch bugs early, or during deployment, to allow only integrity-preserving stored procedures to be accepted. We work in the setting of a functional multi-tier language, where functional code is compiled to SQL that queries and updates a relational database. We use refinement types to track constraints on data and the underlying database. Our analysis uses a refinement-type checker, which relies on recent highly efficient SMT algorithms to check proof obligations. Our method is based on a list-processing semantics for an SQL fragment within the functional language, and is illustrated by a series of examples.

1 Introduction

This paper makes a case for the idea that database integrity should be maintained by static verification of transactional code, rather than by relying on checks at run time. We describe an implementation of this idea for relational databases, where schemas are defined using SQL table descriptions, and updates are written in a functional query language compiled to SQL. Our method relies on a semantics of SQL tables (including constraints) using refinement types, and a semantics of SQL queries in terms of list processing. We describe a series of database schemas, the implementation of transactions in the .NET language F#, and the successful verification of these transactions using the refinement-type checker Stateful F7. Like several recent tools, Stateful F7 relies in part on pushing verification conditions to external SMT solvers, provers whose effectiveness has recently improved at a remarkable rate. Our aim is to initiate the application of modern verification tools for functional languages to the problem of statically-verified database transactions, and to provide some evidence that the idea is at last becoming practical.

1.1 Background: Database Integrity Constraints

SQL table descriptions may include various sorts of constraints, as well as structural information such as base types for columns.

A *check constraint* is an assertion concerning the data within each row of a table, expressed as a Boolean expression.

A *primary key constraint* requires that a particular subset, the *primary key*, of the columns in each row of the table identifies the row uniquely within the table. A key consisting of multiple column labels is called a *composite key*. A *uniqueness* constraint is similar to a primary key constraint but based on a single column (and introduced by the **unique** keyword).

A *foreign key constraint* requires that a particular subset, a *foreign key*, of the columns in each row of the table refers uniquely to a row in the same or anot her table. Satisfaction of primary key and foreign key constraints is known as *referential integrity*.

To illustrate these constraints by example consider a table recording marriages between persons, represented by integer IDs. A key idea is that the marriage of A and B is encoded by including both the tuples (A,B) and (B,A) in the table.

An Example Table with Integrity Constraints: Marriage

```
create table [Marriage](
    [Spouse1] [int] not null unique,
    [Spouse2] [int] not null,
  constraint [PK_Marriage] primary key ([Spouse1],[Spouse2]),
  constraint [FK_Marriage] foreign key ([Spouse2], [Spouse1])
      references [Marriage] ([Spouse1], [Spouse2]),
  constraint [CK_Marriage] check (not([Spouse1] = [Spouse2])))
```

The two columns Spouse1 and Spouse2 in the Marriage table store non-null integers. Database integrity in this example amounts to three constraints: marriage is monogamous (you cannot have two spouses), symmetric (if you marry someone they must be married to you), and irreflexive (you cannot marry yourself).

- The primary key constraint PK_Marriage in conjunction with the uniqueness constraint on the column Spouse1 asserts that nobody is Spouse1 in two different marriages, hence enforcing monogamy.
- The self-referential foreign key constraint FK_Marriage asserts that whenever row (A,B) exists in the table, so does the row (B,A), hence enforcing symmetry.
- The check constraint CK_Marriage asks that nobody is married to themselves, hence enforcing irreflexivity.

A buggy transaction on this table may violate its constraints. The sorts of bugs we aim to detect include the following: (1) insertion of null in Spouse1 or Spouse1 (violating the **not null** type annotation); (2) inserting (A,C) when (A,B) already exists (violating the primary key constraint); (3) inserting (A,B) but omitting to insert (B,A) (violating the foreign key constraint); and (4) inserting (A,A) (violating the check constraint). We aim to eliminate such integrity violations by static analysis.

1.2 Background: Multi-Tier Functional Programming

We consider the common situation where database updates are not written directly in SQL, but instead are generated from a separate programming language via some object-relational mapping. In particular, we consider database transactions expressed in the functional language F# [28], but compiled to SQL for efficient execution in the relational backend. This is an instance of *multi-tier functional programming*, where a single functional program is split across tiers including the web server and the database.

Our mapping is based on three ideas:

(1) We model SQL table definitions as F# types: the whole database is a record type db consisting of named tables, where each table is a list of records, corresponding to the rows of the table.
(2) We provide the user with *standard* functions for create, read, update, and delete operations on each table. We also allow user-supplied *custom* SQL stored procedures, and provide F# functions to call these procedures. Both standard and custom functions are implemented as SQL queries, and can be thought of as imperative actions on a global state of type db.
(3) Users write a transaction as an F# function that interacts with the database by calling a sequence of standard SQL functions and custom stored procedures.

To illustrate point (1), we model our example table definition with the following F# types, where the whole database db is a record with a single field holding the marriages table, which itself is a list of rows.

```
type marriage_row = { m_Spouse1:int; m_Spouse2:int; }
type db = { marriages: marriage_row list; }
```

A row (A,B) is represented by the record:

```
  { m_Spouse1=A; m_Spouse2=B; }
```

The marriage of A and B is represented by the list:

```
[{ m_Spouse1=A; m_Spouse2=B }; { m_Spouse1=B; m_Spouse2=A }]
```

Regarding point (2), we have (among others) the following standard queries as F# functions:

- hasKeyMarriage (A,B) computes whether a row with primary key (A,B) exists in the marriages table.
- deleteMarriagePK (A,B) deletes the row with primary key (A,B) from the marriages table, if it exists.

We have no user-supplied custom SQL queries for the marriages example, but show such queries in some of our later examples.

Actual transactions (point (3) above) are written as functional code. The following example of a user-written transaction is to dissolve a marriage. Given two spouses A and B, we have to check whether the rows (A,B) and (B,A) exist in the database and remove them both.

An Example Transaction: Divorce

```
let divorce_ref (A,B) =
  if hasKeyMarriage(A, B) then
    deleteMarriagePK(A, B);
    deleteMarriagePK(B, A);
    Some(true)
  else Some(false)
```

The body of the function is an expression of type bool option.

- Some **true** means there was a marriage successfully removed, and we commit;
- Some **false** means there was no marriage to remove, and we commit;
- None would mean that the transaction failed and any updates are to be rolled back (a return value not illustrated by this code).

The code above takes care to check that a marriage between A and B already exists before attempting to delete it, and also to remove both (A,B) and (B,A). Instead, careless code might remove (A,B) but not (B,A). Assuming that the foreign key constraint on the marriage table is checked dynamically, such code would lead to an unexpected failure of the transaction. If dynamic checks are not enabled (for instance since the underlying database engine does not support deferred consistency checking) running invalid code would lead to data corruption, perhaps for a considerable duration. Our aim is to detect such failures statically, by verifying the user written code with a refinement-type checker.

1.3 Databases and Refinement Types

The values of a *refinement type* $x{:}T\{C\}$ are the values x of type T such that the formula C holds. (Since the formula C may contain values, refinement types are a particular form of dependent type.) A range of refinement-type checkers has recently been developed for functional languages, including DML [31], SAGE [12], F7 [3], DSolve [23], Fine [27], and Dminor [5], most of which depend on SMT solvers [22].

A central idea in this paper is that refinement types can represent database integrity constraints, and SQL table constraints, in particular. For example, the following types represent our marriage table.

SQL Table Definitions as Refinement Types:

```
type marriage_row = { m_Spouse1:int; m_Spouse2:int }
type marriage_row_ref = m:marriage_row {CK_Marriage(m)}
type marriage_table_ref = marriages:marriage_row_ref list
   { PKtable_Marriage(marriages) ∧ Unique_Marriage_Spouse1(marriages) }
type State = { marriage:marriage_table_ref }
type State_ref = d:State {FK_Constraints(d)}
```

The refinement types use predicate symbols explained informally below. We give formal details later on.

- CK_Marriage(m) means the record m satisfies SQL constraint [CK_Marriage].
- PKtable_Marriage(marriages) means the list of records marriages satisfies the primary key constraint with label [PK_Marriage].
- Unique_Marriage_Spouse1(marriages) means marriages satisfies the SQL uniqueness constraint on column [Spouse1].
- FK_Constraints(d) means the database d satisfies the SQL foreign key constraint with label [FK_Marriage].

1.4 Transactions and the Refined State Monad

The state monad is a programming idiom for embedding imperative actions within functional programs [29]. Pure functions of type $\mathsf{State} \to T * \mathsf{State}$ represent computations that interact with a global state; they map an input state to a result paired with an output state. The *refined state monad* $[(s_0)C_0]\,x{:}T\,[(s_1)C_1]$ is the enrichment of the state monad with refinement types as follows:

$$[(s_0)C_0]\,x{:}T\,[(s_1)C_1] \triangleq s_0{:}\mathsf{State}\{C_0\} \to x{:}T * s_1{:}\mathsf{State}\{C_1\}$$

The formula C_0 is a pre-condition on input state s_0, while the formula C_1 is a post-condition on the result x and output state s_1.

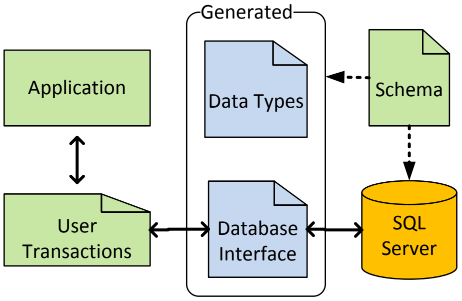
A new idea in the paper is to represent SQL queries and transactions as computations in a refined state monad, with the refinement type State being a record with a field for each table in the database, as above. For example, the function divorce_ref has the following type, where the result of the function is a computation in the refined state monad.

```
val divorce_ref: (int×int) →
  {(s) FK_Constraints(s)} r:bool option {(t) r ≠ None ⇒ FK_Constraints(t)}
```

The return type states that if the function is called in a state s satisfying the foreign key constraints, and it terminates, then it returns a value r of type bool option. Moreover, if r≠None, then the state t after the computation terminates satisfies the foreign key constraints. The type reflects that the code performs sufficient dynamic checks that it never causes a dynamic failure, and that it returns None whenever it leaves the database in an inconsistent state. The case of the function returning None is caught by a transaction wrapper (not shown here) which then aborts the transaction, rolling back the database to its initial state. Buggy code that removes say (A,B) but not (B,A) is caught by type-checking, as it does not re-establish the foreign key constraint FK_Constraints(t).

1.5 An Architecture for Verified Database Transactions

We verify a series of example user transactions, according to the diagram below. Each example starts from a database schema in SQL. From the schema our tool generates refinement types to model the database, and also a functional programming interface for a set of pre-packaged stored procedures in SQL (for actions such as querying and deleting items by key, exemplified by the functions lookupMarriagePK etc mentioned above). Against this interface, the user writes transactional code (exemplified by the function marry_ref mentioned above) in F#, which is invoked from their application. We verify the code of the user transactions using the typechecker Stateful F7 [6], which implements the refined state monad on top of the typechecker F7 [3]. Additionally, not shown in the diagram, in some examples the schema may also include queries written directly as custom SQL stored procedures; we can also verify these queries by mapping SQL into F#.



Our examples are as follows.

(1) Marriages (see above and Section 3).
We have a single table Marriage(Spouse1, Spouse2) with integrity expressed using SQL table constraints. We describe verifiable transactions to create and delete marriages.

(2) Order processing (see Section 4).
We have tables Orders(OrderID, CustomerID, Name, Address) and Details(OrderID, ProductID, UnitPrice, Quantity) with integrity expressed using primary key, foreign key and check constraints. We show that an addOrder function, which creates an order with a single detail row, respects the primary key, foreign key and check constraints.

(3) Heap data structure (see Section 5).
We have a table Heap(HeapID, Parent, Content) where each row represents a node in a heap data structure. Integrity is expressed with SQL constraints plus user-defined constraints written in first-order logic (and not expressible using SQL). We verify that integrity is preserved by recursive functions to push and pop elements, which make use of user-defined stored procedures getRoot and getMinChild.

1.6 Contributions of the Paper

Our main contribution is to interpret SQL table descriptions as refinement types, and database updates as functional programs in the refined state monad, so as to verify, by refinement-type checking, that updates preserve database integrity. Hence, verification of the F# and SQL source code proceeds by sending a series of verification conditions in first-order logic to an automatic theorem prover.

Our source code is in the .NET language F#, but our method could be recast for other functional multi-tier languages, such as Links [9], HOP [25], or FLAPJAX [16], and also for object-oriented programming models such as LINQ [18]. We use the type-checker Stateful F7, but we expect our approach to queries and transactions would easily adapt to related verifiers for functional code with state such as Why [11] or YNOT [19], and indeed to verifiers for imperative code, such as those using Boogie [1].

The idea of static verification of database transactions goes back to the 1970s, to work on computing the weakest precondition needed for a transaction to succeed [14, 7, 26, 4]. Theorem proving technology has improved considerably since the idea of static verification of database transactions was first mooted, and an implication of our work is that the idea is at last becoming practical. Moreover, the success of languages with functional features such as F#, Scala [20], and indeed recent versions of C# with closures, is compelling evidence for the significance of functional programming as an object-oriented technology. Hence, our work lays a foundation for statically verifiable database access from mainstream object-oriented platforms.

Additional details are available at `http://johannes.borgstroem.org/drafts/integrityTR.pdf`.

2 A Tool to Model SQL with Refinement Types

This section fleshes out the architecture diagram of our system.

Section 2.1 describes the details of the SQL schemas input by our system, including both the data definition part defining the structure of tables, and also the data manipulation part of queries invoked from stored procedures.

Section 2.2 details how our tool generates data types and a database interface from a schema. The database interface consists of a set of F# functions with types, including preconditions and postconditions, specified in the syntax of Stateful F7. When generating the database interface, our tool automatically includes functions to access a set of standard queries, as well as functions to access any custom stored procedures included in the schema.

Section 2.3 gives a symbolic reference implementation for the generated database interface. The symbolic implementation relies on list processing in a similar fashion to Peyton Jones and Wadler [21], serves as a formal semantics for the interface, and can be typechecked using Stateful F7. We trust, but do not formally verify, that the behaviour of the symbolic implementation corresponds to our concrete implementation in terms of sending queries to an actual SQL database.

Finally, Section 2.4 extends our schema syntax with the ability to write integrity constraints directly as first-order predicates.

2.1 SQL Schemas: Tables and Stored Procedures

Let c range over constants, x over variables and f,g over table column names. Then, boolean expressions and value expressions occurring within SQL queries are defined by the syntax below. Value expressions include (boolean, integer, and string) constants, binary operations, variables and table field names. Boolean expressions include equations between value expressions, comparisons, conjunction, disjunction and negation.

Values and Expressions

B	$::= E = E \mid E \odot E \mid B \vee B \mid B \wedge B \mid \neg B$	Boolean expression
$\odot$	$::= < \mid <= \mid > \mid >=$	Comparison operator
E	$::= E \oplus E \mid c \mid x \mid f$	Value expression
$\oplus$	$::= + \mid - \mid * \mid /$	Binary operator

A table declaration defines a table t, and gives a name f_i and type T_i to each of its columns. To indicate uniqueness constraints, each column has a tag u_i, either **unique** or empty. SQL supports several data types; in this work, we only consider Boolean, Int and String data types which are defined as their counterparts in F#, and we interpret more complex types using appropriate encodings over the three basic types. We assume that each table has exactly one primary key, which may be composite, exactly one check constraint, and no multiple foreign key references to the same table. (SQL syntax allows multiple check constraints, but these may be conjoined to produce a single check constraint.) Additionally

Data Definition

DT ::=	Table declaration
table t $(u_i\ f_i : T_i)^{i \in 1..n}$,	name and fields
primary key $\overline{g}$,	primary key
check B,	check constraint
$\kappa_1, \ldots, \kappa_m$	foreign keys
κ ::= **foreign key** $\overline{f}$ **references** $t'(\overline{g})$	
T ::= Boolean | Int | String	Type

The syntax of supported SQL queries includes those necessary for selecting, inserting and deleting rows from a table. Let t denote an SQL table name and let $\overline{f}$ be a shorthand for $f_1, \ldots, f_n$ (all the columns of the table) and $\overline{g}$ be a shorthand for $g_1, \ldots, g_m$ (denoting some of the f_i).

Data Manipulation

Q ::= QS | QI | QD | QU	Query
QS ::=	Select query
select [**top** 1] $\overline{g}$	selector
from t	source
where B	criterion
[**order by** f {**asc** | **desc**}]	ordering
QI ::=	Insert query
insert into t	target table
$(f_1, \ldots, f_n)$	table fields
values $(E_1, \ldots, E_n)$	field values
QD ::=	Delete query
delete from t	target table
where B	criterion
QU ::=	Update query
update t **set**	target table
$(g_1, \ldots, g_m)$ =	table fields
$(E_1, \ldots, E_m)$	field values
where B	criterion

A QS query filters all the rows of a table t, based on a boolean criterion B, and projects the selected fields $\overline{g}$. The result of a select query is a table of rows matching the criterion. We consider only **select** queries that contain **top** 1 if and only if they contain an

order by clause. In this case, the resulting table is ordered in either ascending or descending order based on the single field f, and the first element of the table is returned as the result. A *QI* query adds a row consisting of the values $E_1, \ldots, E_n$ in table t. In **insert into** $t\ (f_1, \ldots, f_n)$ **values** $(E_1, \ldots, E_n)$, we expect each E_i to be either a variable or a constant. The result of a *QI* query is a number indicating the count of successful insertions. A *QD* query removes from table t all the rows matching the boolean criterion B. Again, the result is a number indicating the count of successful deletions. Finally, a *QU* query modifies fields $\overline{g}$ to contain values $\overline{E}$ for all rows of table t that match the boolean condition B.

The SQL schema syntax includes constructs for databases, tables, procedures and constraints. A schema is a named tuple of declarations. A declaration can be either a procedure or a table. A procedure abstracts a query Q by giving a name h and parametrises it through the arguments $a_1, \ldots, a_n$. We assume that in a procedure declaration, the query Q only contains variables from $\overline{a}$.

SQL Schema

S	$::=$	$\textbf{schema}\ s(DT_i^{\,i\in 1..n}, DP_j^{\,j\in 1..m})$	Schema
DP	$::=$	$\textbf{procedure}\ h\ (a_i : T_i)^{i\in 1..n} Q$	Procedure declaration

In subsequent sections, we adopt a convenient syntax for advanced queries and assume standard encodings of these syntactic forms in terms of the core query syntax. For example, multi-row insertion is defined in terms of multiple single-row insertions and the star (*) syntax in *QS* queries corresponds to explicitly naming all the columns in the table, in order of their appearance in the table declaration.

2.2 Generating Types and Database Interfaces from Schemas

Our tool maps an SQL schema S to a Stateful F7 module $[\![S]\!]$ by assembling a series of type definitions, predicate definitions, and function signatures. This section describes each of the five components in turn.

Translation from S in SQL to $[\![S]\!]$ in Stateful F7:

Let $[\![S]\!]$ be the Stateful F7 module obtained from schema S by concatenating the type and function definitions displayed below: (S1) types from schema declarations; (S2) refinement formulas from constraints; (S3) signatures of standard functions; (S4) signatures of custom functions.

First, we fix a type for table declarations and the global type `State` of the refined state monad used by Stateful F7. Second, we define logical axioms which correspond to the database constraints. Third we give types to queries and procedures that manipulate the global state. As discussed earlier, expressions get *computation types* of the form $[(s_0)C_0]\,x{:}T\,[(s_1)C_1]$. Finally we generate standard API functions for manipulating the global state. We use **val** `f` : `T` to give a type to a function in the API.

We assume a fixed schema s defined by S. For every table t in s, assume the definition **table** $t\ (u_i\ f_i : T_i)^{i\in 1..n}$, **primary key** $\overline{g}$, **check** $B, \kappa_1, \ldots, \kappa_l$. Given table t, the translation algorithm works as follows: We generate the type `t_key` as a tuple of the corresponding types of the primary key fields and let `Tf`, the type of field `f`, be given by T_i

when $f = f_i$. For each row in the table we create an unrefined record type t_row with the labels corresponding to the column names from the table definition. To associate a **check** constraint with each table row, we refine the row type with the formula CK_t(row) (defined below) and create the refinement type t_row_ref. Values of this type represent rows in the table t for which the check constraint holds. The table t itself is modelled as a list of refined rows (t_table). Finally, we refine the table type by associating the primary key constraint formula PKtable_t(tab) (defined below) with it. Values of this type represent tables for which the primary key constraint holds. Basic types translate directly to their equivalents in Stateful F7. We deal with **not null** and **null** constraints by declaring nullable types as option types.

We can now proceed to the definition of the type corresponding to the database (for a single schema). Without loss of generality, assume that the tables $t_1, \ldots, t_n$ belong to the fixed schema s. The database type is a record of refined tables. In the refined state type, the refinement asserts that values of this type will satisfy the foreign key constraints on the database. The normal state type does not have this refinement, denoting that top-level constraints may temporarily be invalidated. A valid transaction may assume that the foreign key constraints hold, and must enforce them on exit, but may internally temporarily violate the constraints.

(S1) Types from Table Declarations

```
type t_key = Tg1× ... ×Tgn
type t_row = {f1:T1; ...; f n:Tn}
type t_row_ref = row:t_row {CK_t(row)}
type t_table_ref = tab:t_row_ref list {PKtable_t(tab) ∧ ⋀_{u_i=UNIQUE} Unique_t_fi}
type State = { (t_i : ti_table_ref) ^{i∈1..n} }
```

Here check constraints are written as refinements on the row type, while primary key constraints are refinements on the table type. Finally, foreign key constraints are written as refinements on the database type.

We now define logical predicates corresponding to SQL table constraints. We assume a translation $[\![\cdot]\!]^{\mathsf{L}}_r$ from SQL boolean and value expressions to logical formulas and terms with the function; the translation is homomorphic except for the base case $[\![f]\!]^{\mathsf{L}}_r \triangleq r.f$.

- CK_t(row) means the check constraint of table t holds of the tuple row.
- PK_t(r,k) means the primary key of row r of table t is k.
- PKtable_t(tab) means the contents tab of table t satisfies its primary key constraint.
- Unique_t_f(tab) means the contents tab of table t satisfies the uniqueness constraint for field f.
- FK_t_u(tab1,tab2) means the contents tab1 of table t satisfies the foreign key constraint with reference to the contents tab2 of table u.
- FK_Constraints(db) means all foreign key constraints in the database db are satisfied.

In the table below, the Stateful F7 keyword **assume** introduces a universally quantified formula to define each new predicate symbol.

(S2) Refinement Formulas from Constraints

assume $\forall$row. CK_t(row) $\Leftrightarrow [\![B]\!]^{\mathrm{L}}_{row}$

assume $\forall$row,$\bar{x}$. PK_t(row,($\bar{x}$)) $\Leftrightarrow \bigwedge_i x_i$ = row.g_i

assume $\forall$tab. PKtable_t(tab) $\Leftrightarrow\forall$row1, row2.

  (Mem(row1,tab) $\wedge$ Mem(row2,tab) $\wedge$ PK_t(row1,(row2.$g_1, \ldots,$ row2.g_m))) $\Rightarrow$ row1 = row2

assume $\forall$tab. Unique_t_f(tab) $\Leftrightarrow\forall$row1, row2.

  (Mem(row1,tab) $\wedge$ Mem(row2,tab) $\wedge$ row1.f = row2.f) $\Rightarrow$ row1 = row2

assume $\forall$t,u. FK_t_u(t,u) $\Leftrightarrow\forall$x. Mem(x,t) $\Rightarrow$ ($\exists$ y. Mem(y,u) $\wedge \bigwedge_i x.f_i = y.g_i$)

  if $\exists\kappa_i =$ **foreign key** $f_1 \ldots f_m$ **references** $u(g_1 \ldots g_m)$

assume $\forall$s. FK_Constraints(s) $\Leftrightarrow \bigwedge_{t,u}$ FK_t_u(s.t, s.u)

Now that we have translated SQL data declarations, we may proceed to the query and data manipulation languages. A simple select query does not modify the state, and returns a list whose elements are exactly the rows in the table matching the select condition. A select **top** 1 query also does not modify the state, and returns a list which is either empty or contains one element from the table that matches the select criterion and is greater than (or less than, not shown) any other such element. An insert query may only be called if inserting the row does not invalidate any table constraints, and the new table after running the query is the old table with the inserted row prepended to it. A delete query modifies the corresponding table to contain only those rows not matching the query condition. An update query also modifies the table to contain exactly those rows that do not match the where clause, or the updated version of the rows that do.

Types of SQL Queries

$\mathcal{T}[\![$**select** $\overline{g}$ **from** t **where** $B\ ]\!]\triangleq$

  [(s)] l:T$_{\overline{g}}$ list [(s') s=s' $\wedge$($\forall$ x. Mem(x,l) $\Leftrightarrow\exists$r. $\wedge [\![B]\!]^{\mathrm{L}}_r \wedge \overline{r.g} = \overline{x.g} \wedge$ Mem(r,s.t))]

$\mathcal{T}[\![$**select top** 1 $\overline{g}$ **from** t **where** B **order by** f **asc** $]\!]\triangleq$

  [(s)] l:T$_{\overline{g}}$ list [(s') s=s' $\wedge$((l = [] $\wedge$($\forall$r. Mem(r,s.t) $\Rightarrow\neg[\![B]\!]^{\mathrm{L}}_r$)) $\vee$

    ($\exists$x. $[\![B]\!]^{\mathrm{L}}_x \wedge$ Mem(r,s.t) $\wedge$ l = [{$\overline{g} = \overline{r.g}$}] $\wedge$($\forall$r. $[\![B]\!]^{\mathrm{L}}_r \wedge$ Mem(r,s.t) $\Rightarrow$ r.f >= x.f)))]

$\mathcal{T}[\![$**insert into** $t\ (f_1, \ldots, f_n)$ **values** $(E_1, \ldots, E_n)\ ]\!]\triangleq$

  [(s) TC_t({$\overline{f} = \overline{E}$}::(s.t))] unit [(s') s= {s' **with** t = {$\overline{f} = \overline{E}$}::(s.t) }]

$\mathcal{T}[\![$**delete from** t **where** $B\ ]\!]\triangleq$

  [(s)] int [(s') $\exists$t'. s'= {s **with** t = t'} $\wedge$($\forall$r. Mem(r,t') $\Leftrightarrow [\![\neg B]\!]^{\mathrm{L}}_r \wedge$ Mem(r,s.t))]

$\mathcal{T}[\![$**update** t **set** $\overline{g} = \overline{E}$ **where** $B\ ]\!]\triangleq$

  [(s)] int [(s') $\exists$t'. s'= {s' **with** t = t'} $\wedge$($\forall$x. Mem(x,t') $\Leftrightarrow$(Mem(x,s.t) $\wedge\neg[\![B]\!]^{\mathrm{L}}_r$) $\vee$

   ($\exists$r.Mem(r,s.t) $\wedge [\![B]\!]^{\mathrm{L}}_r \wedge$ x= {r **with** $\overline{g} = [\![\overline{E}]\!]^{\mathrm{L}}_r$}))]

The standard API defines functions that look up the existence of keys inside a table, generate fresh keys for a table, checks that an unrefined row satisfies the constraints, inserts a refined row in a table, deletes a row from a table and updates a row in a table. The function fresh_t is only defined when the primary key is non-composite, i.e. constraint contains a single field f, and indeed is an integer. The lookup_t function takes a numeric key and returns true if it exists inside the table; the fresh_t function generates a new key that does not exist inside the table; the check_t function takes an unrefined row and makes sure that all the check constraints are satisfied for it; the insert_t function takes a refined row to be inserted in a table and starting from a state that satisfies all the primary key and foreign key constraints performs the insertion; the delete_t function takes a key for a table and removes the associated row from the table.

(S3) Signatures of Standard Functions

```
PKfresh_t(tab, (x̄ )) ≜ ∀r. Mem(r, tab) ⇒ ⋁_i(x_i ≠ r.g_i)
PKexists_t(tab, (x̄ )) ≜ ∃r. Mem(r, tab) ∧ ⋀_i(x_i = r.g_i)

val hasKey_t: k:t_key →
    [(s) True] b:bool [(s') s=s' ∧(b=false ⇒ PKfresh_t(s.t,k)) ∧(b=true ⇒ PKexists_t(s.t,k))]
val lookup_t: k:t_key →
     [(s) True] o:t_row_ref option [(s') s=s' ∧(o=None ⇒ PKfresh_t(s.t,k)) ∧
         (∀r. o=Some(r) ⇒ Mem(r,s.t) ∧ PK_t(r,k))]
val fresh_t: unit → [(s) True] k:t_key [(s') s=s' ∧ PKfresh_t(s.t,k)]
val check_t: r:t_row → [(s) True] b:bool [(s') s=s' ∧(b=true ⇒ CK_t(r))]
val insert_t: r:t_row_ref → [(s) True] b:bool [(s') (b=false ⇒ s=s') ∧(b=true ⇒ ...)]
val update_t: r:t_row_ref → [(s) True] b:bool [(s') (b=false ⇒ s=s') ∧(b=true ⇒ ...)]
val delete_t: k:t_key → [(s) True] unit [(s') (b=false ⇒ s=s') ∧(b=true ⇒ ...)]
```

To complete the four parts of the definition of $[\![S]\!]$, each custom stored procedure explicitly listed in the schema S is translated to a function signature as follows.

(S4) Signatures of Custom Functions

```
[[procedure h (a_i : T_i)^{i∈1..n} Q ]]≜
  val h : a_1 : 𝔗[[T_1 ]]→ ... → a_n : 𝔗[[T_n ]]→ 𝔗[[Q ]]
```

2.3 Reference Implementation of Database Interface

The dynamic semantics for the subset of SQL that we consider follows Peyton Jones and Wadler [21]. In the following, we assume standard map and filter functions on lists, and also functions max and min that select the maximum and minimum of a list of orderable values. As a convention, we use $\overline{f}$ for the full tuple of columns, and $\overline{g}$ for a subset of the columns. The variable s in the translation represents the entire database record, and therefore the expression s.t projects the table t over which the query is performed. The translation $[\![\cdot]\!]^{\mathrm{F}}_{r}$, from SQL boolean and value expressions to F7 expressions is homomorphic except for the base case $[\![f]\!]^{\mathrm{F}}_{r} \triangleq r.f$.

Semantics of SQL Queries

```
[[select ḡ from t where B]] ≜
  let s = get() in map (fun f̄ → ḡ)(filter (fun r → [[B]]^F_r) (s.t))
[[select top 1 ḡ from t where B order by f asc]] ≜
  match [[select f from t where B]] with [] → [] | xs →
  let m = max(xs) in [hd([[select ḡ from t where (f = m) ∧ B]])]
[[insert into t (f_1, ..., f_n) values (E_1, ..., E_n)]] ≜
  let s = get() in set {s with t = {f_1 = E_1, ..., f_n = E_n} :: (s.t)}
[[delete from t where B]] ≜
  let s = get() in set {s with t = [[select ∗ from t where ¬B]]}
```

```
$[\![\textbf{update}\ t\ \textbf{set}\ \overline{g} = \overline{E}\ \textbf{where}\ B]\!] \triangleq$
    let $s = \textbf{get}()$ in
    let $t1 = [\![\textbf{select}\ *\ \textbf{from}\ t\ \textbf{where}\ \neg B]\!]$ in
    let $tB = [\![\textbf{select}\ *\ \textbf{from}\ t\ \textbf{where}\ B]\!]$ in
    let $t2 = \mathsf{map}\ (\textbf{fun}\ r \rightarrow \{r\ \textbf{with}\ \overline{[\![g]\!]^{\mathrm{F}}_r = [\![E]\!]^{\mathrm{F}}_r}\})\ tB$ in
    set $\{s\ \textbf{with}\ t = t1@t2\}$
```

To translate a simple *QS* query, we first obtain the current database and project the table *t* we are interested in. We then filter every row *r* using the translation of the boolean condition *B*. Finally, we map a projection function, which selects the required subset of columns $\overline{g}$, onto the filtered result. The translation of a *QS* query with **top** and **order by** first narrows the result set using the boolean criterion *B*. If no rows match the criterion, we simply return the empty list. If multiple rows match the criterion, we find the maximum value of the field *f* within any row and store that to a temporary variable *m*. We, finally, use a simple *QS* query to find all the rows that satisfy *B* for which the field *f* has the value *m* and return the head of the list. The translation of a *QI* query involves getting the current database value, and immediately writing it back with the new row being prepended to the existing table. The translation of a *QD* query follows a similar pattern; we get the current database and immediately write back a table consisting of all the row that do not match the boolean condition. The translation of a *QU* query first saves the initial state, as well as the rows of table t that do not match the criterion B. We then extract the rows that match the criterion B, and map the update over them. Finally, the modified state is written back.

Here is the semantics for the API in F#, with appeal to our semantics of SQL in F#. Below we write $pk(\mathsf{t})$ for the non-empty tuple of field names making up the primary key of table t, and we write $ck(\mathsf{t})$ for the check constraint of table t.

Semantics of API Functions

```
let hasKey_t k = [[select * from t where pk(t)=k]] ≠ []
let lookup_t k =
    match [[select * from t where pk(t)=k]] with
   | [r] → Some r
   | _ → None
let fresh_t () =
    genKey_t [[select top 1 pk(t) from t where true order by pk(t) asc]]
let check_t r = $[\![ck(t)]\!]^{\mathrm{F}}_r$
let insert_t r= [[insert into t (f1,...,fn) values (r.f1,...,r.fn)]]=1
let delete_t k = let n = [[delete from t where pk(t)=k]] in ()
let update_t k r = (delete_t k; insert_t r)
```

The user code is written in F# and can be executed symbolically against the reference implementation of the database access API above. The same user code is type-checked against the F7 interface and linked against the concrete implementation of the API functions that use a relational database.

2.4 Extension with Application Constraints

We extend the SQL table syntax from Section 2.1 in order to allow user-specified invariants, written in first-order logic. We also replace the definition of refined tables, and the definition of the global database constraint FK_Constraints as follows.

User Constraints

$\kappa \quad ::= \ \cdots \mid p \qquad\qquad p$ is a unary predicate symbol

$D \quad ::= \ \cdots \mid p \qquad\qquad p$ is a unary predicate symbol

type t_table_ref = tab:t_table$\{$PKtable_t(tab) $\wedge \bigwedge_{i=1\ldots k} p_i^{\mathsf{t}}(tab)\}$

assume $\forall$db. FK_Constraints(db)$\Leftrightarrow \bigwedge_{t,u}$ FK_t_u(s.t, s.u)$\wedge \bigwedge_i p_i^D$

User constraints $p(x)$, where x will be instantiated either by a table or the entire database, are defined by a user-specified first order logic formula C_p that can contain boolean expressions, quantifiers, and other axiomatized predicates. When defining these formulas, care must be taken to avoid introducing inconsistencies—any program satisfies an inconsistent specification.

3 Completing the Marriages Example

Our goal is to type-check application code that accesses the database and to ensure that it respects the database constraints. To achieve this we need a model of the database, the tables and the constraints inside the host language of the application. Based on the rules from section 2.2 and we translate the Marriages table declaration from section 1.1 and generate the appropriate F7 data types with refinements. We now give the complete translation of the marriage example.

3.1 Database Schema

We here repeat the definition of the refined data types corresponding to the marriage table and its rows.

Marriage Data Types

```
type marriage_row = { m_Spouse1:int; m_Spouse2:int }
type marriage_row_ref = m:marriage_row {CK_Marriage(m)}
type marriage_table_ref = marriages:marriage_row_ref list
  { PKtable_Marriage(marriages) ∧ Unique_Marriage_Spouse1(marriages) }
type State = { marriage:marriage_table_ref }
type State_ref = d:State {FK_Constraints(d)}
```

The check constraint, primary key constraint, uniqueness constraint and foreign key constraint are defined as first-order logical formulas, using the keyword **assume**. We define two auxiliary predicates: PK_Marriage(m, k) states that the primary key of row m is k, and FK_Constraints(d) states that all foreign key constraints (of which there is only one) are satisfied for the database d. The predicate Mem(r,t) checks if row r is present in table t.

SQL Constraints as Formulas

```
assume ∀x,y. CK_Marriage((x, y)) ⇔ x ≠ y
assume ∀m,k. PK_Marriage(m, k) ⇔ k = (m.m_Spouse1, m.m_Spouse2)
assume ∀xs. PKtable_Marriage(xs) ⇔
    ∀x,m. Mem(x, xs) ∧ Mem(m, xs) ∧ PK_Marriage(x, (m.m_Spouse1, m.m_Spouse2))
          ⇒ x = m
assume ∀l. Unique_Marriage_Spouse1(l) ⇔
    ∀x,y. Mem(x, l) ∧ Mem(y, l) ∧ x.m_Spouse1 = y.m_Spouse1 ⇒ x = y
assume ∀d. FK_Constraints(d) ⇔ FK_Marriages_Marriages(d.marriages, d.marriages)
assume ∀marriages', marriages. FK_Marriages_Marriages(marriages, marriages') ⇔
    ∀x. Mem(x, marriages') ⇒
        ∃u. Mem(u, marriages) ∧ PK_Marriage(u, (x.m_Spouse2, x.m_Spouse1))
```

3.2 Access function API

From the database schema, our tool also generates data manipulation functions which carry pre-conditions and post-conditions corresponding to the database constraints on their arguments. We generate two implementations of these functions: one that works on the abstract model, and one that works on the actual SQL server database via ADO.Net.

The following code fragment contains the type signatures of the automatically generated functions for the marriage example.

Specification of API Functions

```
val checkMarriage :
  r:marriage_row → [(s)] b:bool [(s')(s = s' ∧ b = true ⇒ CK_Marriage(r))]

val hasKeyMarriage :
  k:(int × int) → [(s)] b:bool [(s')(
    s = s' ∧
    b = false ⇒ PK_Marriages_Fresh(s.marriages, k) ∧
    b = true ⇒ PK_Marriages_Exists(s.marriages, k))]

val deleteMarriagePK :
  k:(int × int) → [(s) PK_Marriages_Exists(s.marriages, k)] unit [(s')
    ContainsiffNotPKMarriage(s, s', k)]

val insertMarriageRowi :
  r:marriage_row_ref → [(s)] b:bool [(s')(
    b = true ⇒ s'.marriages = r :: s.marriages ∧
    b = false ⇒ s = s')]

assume (∀marriages,spouse1,spouse2. (PK_Marriages_Fresh(marriages, (spouse1, spouse2))
      ⇔ (∀x. (Mem(x, marriages) ⇒ (spouse1 ≠ x.m_Spouse1 ∨ spouse2 ≠ x.m_Spouse2)))))
assume (∀marriages,spouse1,spouse2. (PK_Marriages_Exists(marriages, (spouse1, spouse2))
      ⇔ (∃x. ((Mem(x, marriages) ∧ spouse1 = x.m_Spouse1) ∧ spouse2 = x.m_Spouse2))))
```

3.3 User-Written Transactions

In addition to the divorce transaction seen in section 1, the user also writes a transaction to marry two people. Note that the foreign key constraint (symmetry) is temporarily invalidated between the two row insertions. The verification will ensure that it is properly reestablished at the end of the transaction.

Marriage Transaction

```
let marry_ref (A,B) =
  if hasKeyMarriage(A,B) then Some(false)
  else if A=B then Some(false)
  else
    insertMarriageRowi {m_Spouse1=A; m_Spouse2=B};
    insertMarriageRowi {m_Spouse1=B; m_Spouse2=A};
    Some(true)

let marry m = doTransact marry_ref m
```

The final line above defines a transaction marry by calling the transaction wrapper doTransact, which ensures that transactions that may violate database integrity are rolled back. The marriage transaction, wrapped and unwrapped, and the transaction wrapper, have the following types.

Wrapping Transactions

```
type α transaction = [(s) FK_Constraints(s)] r:α  [(t) FK_Constraints(t)]
type α preTransact = [(s) FK_Constraints(s)] r: α option
      [(t) r ≠ None ⇒ FK_Constraints(t)]

val marry_ref: int×int → bool preTransact
val doTransact: (α → β preTransact) → α → (β  option) transaction
```

A transaction returning type α is a computation, which if run in a state satisfying the foreign key constraints, if it terminates, returns a value of type α in a state that satisfies the foreign key constraints. Similarly, a pretransaction returning type α is a computation, which if run in a state satisfying the foreign key constraints, and terminating with a return value of type α option different from None, preserves the foreign key constraints. To go from a pretransaction, e.g., marry_ref to a transaction, e.g., marry, we use the function doTransact which rolls back the pretransaction if it returns None.

We verify that the user code above has the types given above by refinement type checking; in particular, we get that the functions marry and toTransact divorce_ref preserve database integrity.

4 Example: A Simple E-Commerce Application

In this section, we illustrate our approach in the context of a typical e-commerce web shopping cart. A user can add products to their cart, update the number of products or remove items from their order. Each operation must leave the database in a consistent state satisfying all database contraints. An operation either successfully completes the database transaction leaving the database in a new state, or it aborts the transaction and rolls back all the intermediate modifications, leaving the database in its original state.

We store the shopping cart state across two database tables. The first one (*Orders*) holds order information like customer name and shipping address, while the second one (*Details*) stores specific details about orders, like the codes of the chosen products, their quantities and their price. A row in the *Details* table represents a unique product in an order. The column OrderID is used to associate each order with multiple detail rows.

The following SQL fragment shows the two tables, and defines constraints that must hold for the database.

SQL Schema

```
create table [Ordr](
    [OrderID] [int] not null,
    [CustomerID] [nchar](8) null,
    [ShipName] [nvarchar](40) null,
    [ShipAddress] [nvarchar](60) null,
  constraint [PK_Order] primary key ([OrderID])
)
create table [Detail](
    [OrderID] [int] not null,
    [ProductID] [int] not null,
    [UnitPrice] [money] not null,
    [Quantity] [smallint] not null,
  constraint [PK_Detail] primary key ([OrderID], [ProductID]),
  constraint [FK_Details_Orders] foreign key([OrderID])
    references [Ordr] ([OrderID]),
  constraint [CK_Quantity] check (([Quantity]>(0))),
  constraint [CK_UnitPrice] check (([UnitPrice]>=(0))))
```

The primary key is the compound key created from the OrderID and the ProductID fields. To ensure referential integrity, we add the constraint that for every row in the *Details* table, the value of the OrderID field must exist in a row of the *Orders* table. For data integrity, we ask that for each row in the table, Quantity and UnitPrice are non-negative.

E-Commerce Data Types (partial)

```
type State = { ordr : ordr_ref; detail : detail_ref}
type State_ref=d:db{ FK_Constraints(d) }
assume ∀d. FK_Constraints(d) ⇔
  FK_Detail_Ordr(d.detail, d.ordr)
assume ∀ds, os.FK_Detail_Ordr(ds, os) ⇔
  ∀x. Mem(x,ds) ⇒ ∃u.Mem(u,os) ∧ PK_Ordr(u,x.d_OrderID)
```

Given the types corresponding to table definitions (omitted), we represent a database as a record whose labels correspond to the table names. The label types are the refined table types ordr_ref and detail_ref. A refined state State_ref, is a database for which the foreign key constraint between the tables holds. The foreign key predicate definition says that for every row x of the details table, there exists a row u in the orders table, such that the primary key of u is equal to the d_OrderID field of x.

We verify the user defined transaction addOrder below, which takes the necessary data items as arguments and returns a boolean value indicating whether the operation was successful or not.

E-Commerce Transaction

```
let addOrder_ref ord =
  let (customerID, shipName, shipAddress, productID, unitPrice, quantity) = ord in
  let oid = freshOID () in
  let ordr : order_row =
    {o_OrderID = oid; o_CustomerID = customerID;
     o_ShipName = shipName; o_ShipAddress = shipAddress} in
  let detail : detail_row =
    {d_OrderID = oid; d_ProductID = productID;
     d_UnitPrice = unitPrice; d_Quantity = quantity} in
  let x3 = checkDetail detail in
  if x3 then let r = insertDetailRowi detail in
    if r then let r' = insertOrderRowi ordr in
      if r' then true
      else false
    else false
  else false

let addOrder ord = doTransact addOrder_ref ord
```

User defined transactions consist of two parts; a function which when executed may violate database integrity, and a corresponding function that wraps the former one and ensures that the database integrity is reestablished at the end of the transaction, perhaps through a rollback. In the example function addOrder_ref, since the detail is inserted before the order row, the database passes through a state in which the foreign key constraint is violated; so this code would fail needlessly in some systems, such as SQL Server. The function addOrder uses the library function doTransact to wrap addOrder_ref with the necessary transaction handling code.

5 Example: A Heap-Ordered Tree

This example shows the use of more advanced features of our system, such as user-defined predicates and custom stored procedures. We use a database table to store a heap-ordered tree, where the child nodes store pointers to their parent but not vice versa. We add two named application-level invariants (Section 2.4) to the heap table. TR_isHeap states that the value stored at every node is greater than that stored at its parent; and TR_uniqueRoot states that any two root nodes are equal.

Heap SQL Specification

```
create table [Heap](
    [HeapID] [int] identity (1,1) not null,
    [Parent] [int] not null,
    [Content] [int] not null,
  constraint
    [PK_Heap] primary key CLUSTERED ([HeapID] asc),
  constraint
    [FK_Heap] foreign key ([Parent]) references [Heap] ([HeapID]),
  /×−−− UserConstraint TR_isHeap ×/
  /×−−− UserConstraint TR_uniqueRoot ×/)
```

The first-order formulas expressing the application-level invariant predicates are defined in terms of an auxiliary predicate TR_isRoot. This predicate denotes that a given node is the root of a tree, which is defined as the node being its own parent.

User Constraints

```
assume ∀d. TR_isHeap(d) ⇔ (∀x,y. (Mem(x, d) ∧ Mem(y,d) ∧
           x.h_Parent = y.h_HeapID) ⇒ x.h_Content >= y.h_Content)
assume ∀d. TR_uniqueRoot(d) ⇔
             (∀x,y. (TR_isRoot(x,d) ∧ TR_isRoot(y,d)) ⇒ x = y )
assume ∀x,d. TR_isRoot(x,d) ⇔ Mem(x, d) ∧ x.h_Parent = x.h_HeapID
```

We also define two stored procedures: getRoot returns a root node of the tree, while getMinChild returns the smallest child of a given node.

Custom Stored Procedures

```
create procedure getRoot as
    select top 1 ∗ from Heap
    where HeapID = Parent order by Content asc

create procedure getMinChild @rootID [int] as
    select top 1 ∗ from Heap
    where (Parent = @rootID and HeapID ≠ @rootID)
    order by Content asc
```

The form of these stored procedures is very similar, so we detail the translation of only getRoot. Its post-condition is defined as follows. The function can return two different values: the empty list or a list containing one element. If the function returns the empty list, we learn that there is no root element. If one element was returned, the predicate GetRootResult states that it satisfies the where clause, and is from the table, and the predicate GetRootIsMin states that the returned element is the one with the least value of the elements in the table satisfying the where clause.

getRoot

```
val getRoot : unit → [(s)] l:heap_row list
    [(s') s = s' ∧ GetRootResult(l,s) ∧
    ((l = [] ∧ GetRootNotFound(s)) ∨ (∃x. l = [x]))]
```

```
assume ∀s,x. GetRootNotFound(s) ∧ Mem(x,s.heaps) ⇒
        not (x.h_Parent = x.h_HeapID)
assume ∀s,l,x. (GetRootResult(l,s) ∧ (l = [x])) ⇒
        (x.h_HeapID = x.h_Parent) ∧ Mem(x, s.heaps) ∧
        PK_Heaps_Exists(s.heaps,x.h_HeapID) ∧ GetRootIsMin(x,s)
assume ∀x,s,r. (GetRootIsMin(x,s) ∧ r.h_HeapID = r.h_Parent
                  ∧ Mem(r, s.heaps)) ⇒ x.h_Content >= r.h_Content
```

In this setting, we define two operations. We can insert a node into the tree, using the function pushAt_int, which adds a node with a given value as a child to the nearest ancestor of a given node that has a value less than the value to insert. With pop_int we can pop the smallest node off the table, causing its smallest child to bubble up the tree, recursively. This recursive procedure is called rebalanceHeap.

Specifications of User Functions

```
val pushAt_int: int×int → bool preTransact
val pop_int: unit → int preTransact
val rebalanceHeap: i:int →
    [(s) FK_Constraints(s) ∧ PK_Heaps_Exists(s.heaps,i) ]
    unit [(t) FK_Constraints(t) ]
```

To push an element, we compare it to the root. If it is smaller, it becomes the new root value, otherwise we store it as a child of the root.

Pushing an Element Onto the Heap

```
let rec pushAt_int (i,v) =
  let node = lookupHeapPK i in
  let newID = freshHID () in
  match node with
  | None → None
  | (Some(nodeRow)) →
    let {h_Content=c ; h_HeapID=id; h_Parent=par} = nodeRow in
    if v > c then
      let r = {h_Content = v ; h_HeapID = newID; h_Parent = id} in
      if insertHeapRow r then Some(true) else None
    else
      if hasKeyHeap id then
        if hasKeyHeap par then
          if id = par then
            let nodeRow' = {h_Content=v; h_HeapID=id; h_Parent=par} in
            if updateHeapPK id nodeRow' then
              let r = {h_Content=c; h_HeapID=newID; h_Parent=id} in
              if insertHeapRow r then Some(true) else None
            else None
          else pushAt_int (id,v)
        else None
      else None
```

When popping the root, we use rebalanceHeap to let a chain of minimal children "bubble up" one step.

Popping the Root of the Heap

```
let rec rebalanceHeap id =
  let minM = getMinChild(id) in match minM with
  | [] → let res = deleteHeapPK id in res
  | [minRow] → match minRow with
    | {h_Content=mc; h_HeapID=mid; h_Parent=mpar} →
      if hasKeyHeap mid then
        let r = lookupHeapPK id in match r with
        | None → ()
        | (Some(u)) → match u with
          | {h_Content=rc ; h_HeapID=rid; h_Parent=rpar} →
            let v = {h_Content = mc; h_HeapID = id ; h_Parent = rpar} in
            updateHeapPK id v;
            let res = rebalanceHeap mid in res
      else ()

let pop_int () =
  let root = getRoot() in match root with
  | [] → None
  | [rootRow] → match rootRow with
    | {h_Content = c; h_HeapID = id; h_Parent = par} →
      (rebalanceHeap id; Some(c))
```

To verify this more complex example, we needed to add three axioms to the context of the SMT solver. The first axiom states that when updating a row, without changing its primary key, then the same primary keys are present in the database table as before. The second axiom states that if the foregn key constraints hold, and the primary and foreign key fields are unchanged by a single-row update, then the foreign key constraints are not violated. The third axiom states that if a row has no children, then it can be deleted without violating the foreign key constraint.

Axioms

```
assume ∀h,h',k,v,x. UpdateHeap(h',h,k,v) ∧ PK_Heaps_Exists(h,x) ⇒ PK_Heaps_Exists(h',x)
assume ∀h1,h2,x,y. FK_Heaps_Heaps(h1,h1) ∧ Replace(h2,h1,x,y) ∧ x.h_Parent = y.h_Parent
     ∧ x.h_HeapID = y.h_HeapID ⇒ FK_Heaps_Heaps(h2,h2)
assume ∀s,k,s'. FK_Constraints(s) ∧ GetMinChildNotFound(k,s) ∧ DeletedHeap(s,s',k) ⇒
     FK_Constraints(s')
```

Given these axioms, we verify that transactions that add values to and/or pop values from the tree do not violate the database integrity, including the application-level constraints.

6 Software Architecture and Evaluation

Our implementation consists of a compiler from an SQL schema to a Stateful F7 database interface implementing the translation in Section 2.2, and from an SQL schema to a symbolic implementation of the database in F# implementing the dynamic semantics of Section 2.3. We use the Stateful F7 typechecker to verify the user supplied transactions against the generated interface. Additionally we provide a concrete implementation of the database interface against SQL Server. The core compiler (without Stateful F7) consists of about 3500 lines of F# code split between the SQL parser, the translation rules and the implementation of the database interface.

We evaluate our approach experimentally by veryfing all the examples of this paper; Table 6 summarizes our results. For each example it gives: a) the total number of lines of user supplied code (this includes the F# transaction code and user-defined predicates, and the SQL schema declaration), b) the number of lines of the automatically generated data types and database interface, and c) the verification information consisting of the number of proof obligations passed to Z3 and the actual verification time. Constraints that affect individual rows or tables like **check**, and **primary key** constraints, unsurprisingly add little time to the verification process. This explains the small verification time of the E-Commerce example, despite having more tables and **check** constraints than the other examples. On the other hand uniqueness, foreign key constraints and arbitrary user constraints require disproportionately more time to verify.

We express constraints in first-order logic with a theory of uninterpreted function symbols and linear arithmetic. The main challenge when working with first-order solver like Z3 is quantifier instantiation. In certain examples like heap, we found that Z3 was unable to prove the automatically generated predicates. As a result and to assist Z3 with its proof obligations, our compiler implements some additional intermediate predicates. In particular, for universally quantified formulas, we gave predicate names to quantified subformulas such that superfluous instantiations might be avoided. For existentially quantified formulas, Z3 sometimes has problems constructing the appropriate witness, and we instead were forced to add an axiom that referred to predicates that abstracted quantified subformulas. One contributing factor to this problem was that F7 communicates with the SMT solver Z3 using the Simplify format, while the more advanced SMT-Lib format would permit us to add sorting of logical variables, patterns to guide quantifier instantiation, and access to the array theory implemented in Z3 for a more efficient modelling of tables as arrays indexed by the primary key.

Given that our objective is to statically verify that transactional code contains enough checks to preserve the database invariants, we found that applying our approach interactively as we developed the transactional code, helped us implement an exhaustive set of checks and made for a pleasant programming experience. Additionally, the standard database API provides sufficent building blocks to write transactional code. At the same time our approach leads to very verbose code which, when verified, explicitly handles any possible transaction outcome.

7 Related Work

The idea of applying program verification to database updates goes back to pioneering work [14, 7] advocating the use of Hoare logic or weakest preconditions to verify transactions.

	User supplied		Generated		Verification	
	transactions	schema	data types	db interface	queries	time
Marriages	38	10	38	48	20	20.890s
E-Commerce	41	23	54	74	16	9.183s
Heap	111	30	54	85	76	80.385s

Table 1. Lines of user supplied and generated code, and verification information

Sheard and Stemple [26] describe a system for verifying database transactions in a dialect of ADA to ensure that if they are run atomically then they obey database constraints. The system uses higher order logic and an adaptation of the automated techniques of Boyer and Moore.

In the setting of object-oriented databases, Benzaken and Doucet [4] propose that the checking procedures invoked by triggers be automatically generated from high-level constraints, well-typed boolean expressions.

Benedikt, Griffin, and Libkin [2] consider the integrity maintenance problem, and study some theoretical properties of the weakest preconditions for a database transaction to succeed, where transactions and queries are specified directly in first-order logic and extensions. Wadler [30] describes a related practical system, Pdiff, for compiling transactions against a large database used to configure the Lucent 5ESS telephone switch. Consistency constraints on a database with nearly a thousand tables are expressed in C. Transactions in a functional language are input to Pdiff, which computes the weakest precondition which must hold to ensure the transaction preserves database integrity.

To the best of our knowledge, our approach to the problem is the first to be driven by concrete SQL table descriptions, or to be based on an interpretation of SQL queries as list processing and SQL constraints as refinement types, or to rely on SMT solvers.

A recent tool [10] analyzes ADO.NET applications (that is, C# programs that generate SQL commands using the ADO.NET libraries) for SQL injection, performance, and integrity vulnerabilities. The only integrity constraints they consider are check constraints (for instance, that a price is greater than zero); they do not consider primary key and foreign key constraints.

Malecha and others [17] use the Coq system to build a fully verified implementation of an in-memory SQL database, which parses SQL concrete syntax into syntax trees, maps to relational algebra, runs an optimizer and eventually a query. Their main concern is to verify the series of optimization steps needed for efficient execution. In contrast, our concern is with bugs in user transactions rather than in the database implementation. Still, our work in F# is not fully verified or certified, so for higher assurance it could be valuable to port our techniques to this system.

Ur/Web [8] is a web programming language with a rich dependent type system. Like our work, Ur/Web has a dependently typed embedding of SQL tables, and can detect typing errors in embedded queries. On the other hand, static checking that transactions preserve integrity is not an objective of the design; Ur/Web programs may result in a “fatal application error if the command fails, for instance, because a data integrity constraint is violated” (online manual, November 2010).

Refinement-type checkers with state are closely related to systems for Extended Static Checking such as ESC Java [13] and its descendants [1]. To the best of our knowledge, these systems have not previously been applied to verification of transactions, but we expect it would be possible.

8 Conclusion

We built a tool for SQL databases to allow transactions to be written in a functional language, and to be verified using an SMT-based refinement-type checker. On the basis of our implementation experience, we conclude that it is feasible to use static verification to tell whether transactions maintain database integrity.

In future, we are interested to consider an alternative architecture in which our static analysis of queries is implemented in the style of proof-carrying code on the SQL server itself. Another potential line of work is to model database state within separation logic, and to appeal to its tools for reasoning about updates.

Acknowledgements Discussions with Peter Buneman, Giorgio Ghelli, and Tim Griffin were useful.

A Stateful F# (Review)

We describe preconditions, postconditions, and refinements for a subset of F#.

We begin with its syntax and operational semantics in Section A.1 and Section A.2. Section A.3 describes the type system of RIF and its soundness with respect to the operational semantics.

Our starting point is the Fixpoint Calculus (FPC) [**?**,15], a deterministic call-by-value λ-calculus with sums, pairs and iso-recursive data structures.

A.1 A Functional Programming Language

Syntax of the Core Fixpoint Calculus:

s,x,y,z	variable
$h ::=$	value constructor
$\quad\mathsf{inl}$	$\quad$left constructor of sum type
$\quad\mathsf{inr}$	$\quad$right constructor of sum type
$\quad\mathsf{fold}$	$\quad$constructor of recursive type
$M,N ::=$	value
$\quad x$	$\quad$variable
$\quad ()$	$\quad$unit
$\quad \mathbf{fun}\,x \to A$	$\quad$function
$\quad (M,N)$	$\quad$pair
$\quad h\,M$	$\quad$construction
$A,B ::=$	expression
$\quad M$	$\quad$value
$\quad M\,N$	$\quad$application
$\quad M = N$	$\quad$syntactic equality
$\quad \mathbf{let}\ x = A\ \mathbf{in}\ B$	$\quad$let
$\quad \mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$	$\quad$pair split
$\quad \mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B$	$\quad$constructor match

We identify all phrases of syntax up to the consistent renaming of bound variables. Here x is bound with scope A in $\mathbf{fun}\,x \to A$, in $\mathbf{match}\ M\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B$, and in $\mathbf{let}\ x = B\ \mathbf{in}\ A$, and x and y are bound with scope A in $\mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$. We write $\overline{M}$ as shorthand for a possibly empty sequence $M_1,\ldots,M_n$, and similarly for $\overline{x}$, $\overline{A}$, etc. We write the empty sequence as $\circ$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{x}$ is written $|\overline{x}|$. If ϕ is a phrase of syntax (such as an expression), we let $fv(\phi)$ and $\mathsf{fn}(\phi)$ be the sets of variables and names occuring free in ϕ. We write $\phi\,\{^{\psi}\!/_{x}\}$ for the outcome of the capture-avoiding substitution of ψ for each free occurrence of x in the phrase ϕ.

A value may be a variable x, the unit value $()$, a function $\mathbf{fun}\,x \to A$, a pair (M,N), or a construction. The constructions $\mathsf{inl}\ M$ and $\mathsf{inr}\ M$ are the two sorts of value of sum type, while the construction $\mathsf{fold}\ M$ is a value of an iso-recursive type.

In our formulation of FPC, the syntax of expressions is in a reduced form in the style of A-normal form [24], where sequential composition of redexes is achieved by inserting suitable let-expressions. The other expressions are function application $M\ N$, equality $M = N$ (which tests whether the values M and N are syntactically identical), pair splitting $\mathbf{let}\ (x,y) = M\ \mathbf{in}\ A$, and constructor matching $\mathbf{match}\ M\ \mathbf{with}\ h\ x \rightarrow A\ \mathbf{else}\ B$.

To complete our calculus, we augment FPC with the following operations for manipulating and writing assertions about a global state. The state is implicit and is simply a value of the calculus. We assume an untyped first-order logic with equality over values, equipped with a *deducibility relation* $S \vdash C$, from finite multisets of formulas to formulas.

Completing the Syntax: Adding Global State

$A, B ::=$	expression
$\cdots$	expressions of the Fixpoint Calculus
$\mathbf{get}()$	get current state
$\mathbf{set}(M)$	set current state
$\mathbf{assume}\ (s)C$	assumption of formula C
$\mathbf{assert}\ (s)C$	assertion of formula C
$C ::=$	formula
$p(M_1, \ldots, M_n)$	atomic formula, p a predicate symbol
$M = M'$	equation
$C \wedge C' \mid \neg C \mid \exists x.C$	standard connectives and quantification

The expression $\mathbf{get}()$ returns the current state as its value. The expression $\mathbf{set}(M)$ updates the current state with the value M and returns the unit value $()$.

We specify intended properties of programs by embedding assertions, which are formulas expected to hold with respect to the *log*, a finite multiset of assumed formulas. The expression $\mathbf{assume}\ (s)C$ adds the formula $C\{^{M}/_{s}\}$ to the logged formulas, where M is the current state, and returns $()$. The expression $\mathbf{assert}\ (s)C$ immediately returns $()$; we say the assertion *succeeds* if the formula $C\{^{M}/_{s}\}$ is deducible from the logged formulas, and otherwise that it *fails*. In both $\mathbf{assert}\ (s)C$ and $\mathbf{assert}\ (s)C$, s is bound with scope C. This style of embedding assumptions and assertions within expressions is in the spirit of the pioneering work of Floyd, Hoare, and Dijkstra on imperative programs; the formal details are simply an imperative extension of assumptions and assertions in RCF [3].

We use some syntactic sugar to make it easier to write and understand examples. We write $A; B$ for $\mathbf{let}\ \_ = A\ \mathbf{in}\ B$. We define boolean values as $\mathbf{true} \triangleq \mathsf{inl}\ ()$ and $\mathbf{false} \triangleq \mathsf{inr}\ ()$. Conditional statements can then be defined as $\mathbf{if}\ M\ \mathbf{then}\ A\ \mathbf{else}\ B \triangleq \mathbf{match}\ M\ \mathbf{with}\ \mathsf{inl}\ x \rightarrow A\ \mathbf{else}\ B$. We write $\mathbf{let\ rec}\ f\ x = A\ \mathbf{in}\ B$ as an abbreviation for defining a recursive function f, where the scope of f is A and B, and the scope of x is A. When s does not occur in C, we simply write C for $(s)C$. In our examples, we often use a more ML-like syntax, lessening the A-normal form restrictions of our calculus. In particular, we use $\mathbf{let}\ f\ x = A$ for $\mathbf{let}\ f = \mathbf{fun}\ x \rightarrow A$, $\mathbf{if}\ A\ \mathbf{then}\ B\ \mathbf{else}\ C$ for $\mathbf{let}\ x = A\ \mathbf{in}\ \mathbf{if}\ x\ \mathbf{then}\ B\ \mathbf{else}\ C$ (where $x \notin fv(B,C)$), $\mathbf{let}\ (x,y) = A\ \mathbf{in}\ B$ for $\mathbf{let}\ z = A\ \mathbf{in}\ \mathbf{let}\ (x,y) = z\ \mathbf{in}\ B$ (where $z \notin fv(B)$), and so on. See [3], for example, for a discussion of how to recover standard functional programming syntax and data types like Booleans and lists within the core Fixpoint Calculus.

A.2 Semantics

We formalize the semantics of our calculus as a small-step reduction relation on configurations, each of which is a triple (A,N,S) consisting of a closed expression A, a state N, and a log S, which is a multiset of formulas generated by assumptions.

The present the rules for reduction in two groups. The first group consists of rules that are independent of the current state, and which correspond to the semantics of core FPC.

Reductions for the Core Calculus: $(A,N,S) \longrightarrow (A',N',S')$

$\mathcal{R} ::= [\,] \mid \mathbf{let}\ x = \mathcal{R}\ \mathbf{in}\ A$ evaluation context

$$(\mathcal{R}[A],N,S) \longrightarrow (\mathcal{R}[A'],N',S') \qquad \text{if } (A,N,S) \longrightarrow (A',N',S')$$
$$((\mathbf{fun}\, x \to A)\ M,N,S) \longrightarrow (A\{^{M}\!/_{x}\},N,S)$$
$$(M_1 = M_2,N,S) \longrightarrow (\mathbf{true},N,S) \qquad \text{if } M_1 = M_2$$
$$(M_1 = M_2,N,S) \longrightarrow (\mathbf{false},N,S) \qquad \text{if } M_1 \neq M_2$$
$$(\mathbf{let}\ x = M\ \mathbf{in}\ A,N,S) \longrightarrow (A\{^{M}\!/_{x}\},N,S)$$
$$(\mathbf{let}\ (x,y) = (M_1,M_2)\ \mathbf{in}\ A,N,S) \longrightarrow (A\{^{M_1}\!/_{x}\}\{^{M_2}\!/_{y}\},N,S)$$
$$(\mathbf{match}\ (h\ M)\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B,N,S) \longrightarrow (A\{^{M}\!/_{x}\},N,S)$$
$$(\mathbf{match}\ (h'\ M)\ \mathbf{with}\ h\ x \to A\ \mathbf{else}\ B,N,S) \longrightarrow (B,N,S) \text{ if } h \neq h'$$

The second group of rules formalize the semantics of the get and set operators, and of assumptions and assertions, described informally in the previous section.

Reductions Related to State: $(A,N,S) \longrightarrow (A',N',S')$

$$(\mathbf{get}(),N,S) \longrightarrow (N,N,S)$$
$$(\mathbf{set}(M),N,S) \longrightarrow ((),M,S)$$
$$(\mathbf{assume}\ (s)C,N,S) \longrightarrow ((),N,S \cup \{C\{^{N}\!/_{s}\}\})$$
$$(\mathbf{assert}\ (s)C,N,S) \longrightarrow ((),N,S)$$

We say an expression is safe if none of its assertions may fail at runtime. A configuration (A,N,S) has *failed* when $A = \mathcal{R}[\mathbf{assert}\ (s)C]$, for some evaluation context $\mathcal{R}$, and we cannot derive $S \vdash C\{^{N}\!/_{s}\}$. A configuration (A,N,S) is *safe* if and only if there is no failed configuration reachable from (A,N,S), that is, for all (A',N',S'), if $(A,N,S) \longrightarrow^{*} (A',N',S')$ then (A',N',S') has not failed.

The purpose of the type system in the next section is to establish safety by typing.

A.3 Types

There are two categories of type: *value types* characterize values, while *computation types* characterize the imperative computations denoted by expressions. Computation types resemble Hoare triples, with preconditions and postconditions.

Syntax of Value Types and Computation Types:

$T,U,V ::=$	value type
α	type variable
unit	unit type

$x{:}T \rightarrow F$	dependent function type
$x{:}T * U$	dependent pair type
$T + U$	disjoint sum type
$\mu\alpha.T$	iso-recursive type (scope of α is T)
$x{:}T\{C\}$	refinement type
$F, G ::=$	expression type
$[(s_0)C_0]\,x{:}T\,[(s_1)C_1]$	

Value types are based on the types of the Fixpoint Calculus, except that function types $x{:}T \rightarrow F$ and pair types $x{:}T * U$ are dependent. A refinement type, $x{:}T\{C\}$, denotes those values x of type T such that C holds. We use these types to describe invariants of the database. In these types, x is bound, with scope F, U and C, respectively. If x is not used, these types degenerate to simple types. In particular, if x is not free in U and F, we write $T * U$ for $x{:}T * U$ and $T \rightarrow F$ for $x{:}T \rightarrow F$.

An expression type, $[(s)C_1]\,x{:}T\,[(s')C_2]$, denotes computations that when started in a state s satisfying C_1, return a value x of type T in a state s' such that C_2 holds. Here s is in scope in C_1, T and C_2, and x and s' are in scope in C_2. As above, we write $[(s_0)C_0]\,T\,[(s_1)C_1]$ for $[(s_0)C_0]\,x{:}T\,[(s_1)C_1]$ when x is not free in C_1.

When we write a type T in a context where a computation type F is expected, we intend T as a shorthand for the computation type $[(s_0)\mathsf{True}]\,T\,[(s_1)s_1 = s_0]$. This is convenient for writing curried functions: $x{:}T \rightarrow y{:}U \rightarrow F$ stands for $x{:}T \rightarrow [(s'_0)\mathsf{True}]\,y{:}U \rightarrow F\,[(s'_1)s'_1 = s'_0]$.

Typing Rules for Expressions:

(Stateful Exp Let)

$$\frac{\begin{array}{l} E \vdash A : [(s_0)C_0]\,x_1{:}T_1\,[(s_1)C_1] \\ E, s_0, x_1 : T_1 \vdash B : [(s_1)C_1]\,x_2{:}T_2\,[(s_2)C_2] \\ \{s_1, x_1\} \cap fv(T_2, C_2) = \varnothing \end{array}}{E \vdash \mathbf{let}\ x_1 = A\ \mathbf{in}\ B : [(s_0)C_0]\,x_2{:}T_2\,[(s_2)C_2]}$$

(Sub Comp)

$$\frac{\begin{array}{l} fv(C_0, C'_0) \subseteq dom(E, s_0) \quad fv(C_1, C'_1) \subseteq dom(E, s_0, x_1, s_1) \\ C'_0 \vdash C_0 \quad E, s_0 \vdash T_1 <: T'_1 \quad (C'_0 \wedge C_1) \vdash C'_1 \end{array}}{E \vdash [(s_0)C_0]\,x_1{:}T_1\,[(s_1)C_1] <: [(s_0)C'_0]\,x_1{:}T'_1\,[(s_1)C'_1]}$$

In a subtype G of an expression type F, we can strengthen the precondition. The postcondition of G must also be weaker than (implied by) the precondition of G together with the postcondition of F. As an example, $T \rightarrow \{(s)C\}U\{(t)C\{^t/_s\}\}$ is a subtype of $T \rightarrow U$ for every C, since $\vdash C \Rightarrow \mathsf{True}$ and $\vdash (C \wedge s = t) \Rightarrow C\{^t/_s\}$.

Assumptions and Assertions:

(Exp Assume)

$$\frac{E, s_0, s_1 \vdash \diamond \quad fv(C) \subseteq dom(E, s_0)}{E \vdash \mathbf{assume}\ (s_0)C : [(s_0)\mathsf{True}]\,\mathsf{unit}\,[(s_1)((s_0 = s_1) \wedge C)]}$$

(Exp Assert)

$$\frac{E, s_0, s_1 \vdash \diamond \quad fv(C) \subseteq dom(E, s_0)}{E \vdash \textbf{assert}\ (s_0)C : [(s_0)C]\,\mathsf{unit}\,[(s_1)s_0 = s_1]}$$

In (Exp Assume), an assumption **assume** $(s)C$ has C as postcondition, and does not modify the state. Dually, in (Exp Assert), an assertion **assert** $(s)C$ has C as precondition.

Rules for State Manipulation:

(Stateful Get)

$$\frac{E, s_0, s_1 \vdash \diamond}{E \vdash \textbf{get}() : [(s_0)\mathsf{True}]\,x_1{:}\mathsf{state}\,[(s_1)x_1 = s_0 \land s_1 = s_0]}$$

(Stateful Set)

$$\frac{E \vdash M : \mathsf{state} \quad E, s_0, s_1 \vdash \diamond}{E \vdash \textbf{set}(M) : [(s_0)\mathsf{True}]\,\mathsf{unit}\,[(s_1)s_1 = M]}$$

In (Stateful Get), the type of **get**$()$ records that the value read is the current state. In (Stateful Set), the postcondition of **set**(M) states that M is the new state. The postcondition of **set**(M) does not mention the initial state. We can recover this information through subtyping, as seen above.

The main result of this section is that a well-typed expression run in a state satisfying its precondition is *safe*, that is, no assertions fail. Using this result, we can implement different type systems for reasoning about stateful computation in the calculus.

Theorem 1 (Safety). *If* $\varnothing \vdash A : [(s)C]\,\_ : T\,[(s')\mathsf{True}]$, $\varnothing \vdash C\{^M/_s\}$ *and* $\varnothing \vdash M : \mathsf{state}$ *then configuration* $(A, M, \varnothing)$ *is safe.*

A.4 Typechecking by Translation

To typecheck a program, we first use the Stateful F7 tool to translate it into a refined state monad. In particular, a computation of type $[(s)C_1]\,x{:}T\,[(s')C_2]$ is translated into a function of type $s{:}(s{:}\mathsf{state}\{C_1\}) \to x{:}T * s'{:}\mathsf{state}\{C_2\}$. The additional state argument is threaded through the computation. We then apply F7, a refinement type checker for functional programs.

B Source code

B.1 Code for the Marriages Example

Marriages

```
create table [Marriage](
    [Spouse1] [int] not null unique,
    [Spouse2] [int] not null,
  constraint [PK_Marriage] primary key ([Spouse1],[Spouse2]),
  constraint [FK_Marriage] foreign key ([Spouse2], [Spouse1])
      references [Marriage] ([Spouse1], [Spouse2]),
  constraint [CK_Marriage] check (not([Spouse1] = [Spouse2])))
```

Marriages transactions

```
let marry_ref (A,B) =
  if hasKeyMarriage(A,B) then Some(false)
  else if A=B then Some(false)
  else
    insertMarriageRowi {m_Spouse1=A; m_Spouse2=B};
    insertMarriageRowi {m_Spouse1=B; m_Spouse2=A};
    Some(true)

let marry m = doTransact marry_ref m
```

B.2 Code for the Orders Example

Web cart

```
create table [Ordr](
    [OrderID] [int] not null,
    [CustomerID] [nchar](8) null,
    [ShipName] [nvarchar](40) null,
    [ShipAddress] [nvarchar](60) null,
  constraint [PK_Order] primary key ([OrderID])
)
create table [Detail](
    [OrderID] [int] not null,
    [ProductID] [int] not null,
    [UnitPrice] [money] not null,
    [Quantity] [smallint] not null,
  constraint [PK_Detail] primary key ([OrderID], [ProductID]),
  constraint [FK_Details_Orders] foreign key([OrderID])
    references [Ordr] ([OrderID]),
  constraint [CK_Quantity] check (([Quantity]>(0))),
  constraint [CK_UnitPrice] check (([UnitPrice]>=(0))))
```

Web cart transactions

```
let addOrder_ref order =
  let (customerID, shipName, shipAddress, productID, unitPrice, quantity) = order in
  let oid = freshOID () in
  if quantity > 0
  then if unitPrice >= 0
  then begin
    let order : ordr_row = (
      {o_OrderID = oid;
  o_CustomerID = Some(customerID);
  o_ShipName = Some(shipName);
  o_ShipAddress = Some(shipAddress)}) in
    let detail : detail_row = (
      {d_OrderID = oid;
  d_ProductID = productID;
  d_UnitPrice = unitPrice;
  d_Quantity = quantity}) in
    if insertDetailRowi detail then
      if insertOrdrRowi order then
        Some(true)
      else None
    else None
  end
  else None
  else None

let addOrder order = doTransact addOrder_ref order
```

B.3 Code for the Heap Example

A heap database

```
create table [Heap](
    [HeapID] [int] identity (1,1) not null,
    [Parent] [int] not null,
    [Content] [int] not null,
  constraint
    [PK_Heap] primary key CLUSTERED ([HeapID] asc),
  constraint
    [FK_Heap] foreign key ([Parent]) references [Heap] ([HeapID]),
  /×−−− UserConstraint TR_isHeap ×/
  /×−−− UserConstraint TR_uniqueRoot ×/)
```

Heap transactions

```
let rec pushAt_int (i,v) =
  let node = lookupHeapPK i in
  let newID = freshHID () in
  match node with
  | None → None
  | (Some(nodeRow)) →
    let {h_Content=c ; h_HeapID=id; h_Parent=par} = nodeRow in
    if v > c then
      let r = {h_Content = v ; h_HeapID = newID; h_Parent = id} in
      if insertHeapRow r then Some(true) else None
    else
      if hasKeyHeap id then
        if hasKeyHeap par then
          if id = par then
            let nodeRow' = {h_Content=v; h_HeapID=id; h_Parent=par} in
            if updateHeapPK id nodeRow' then
              let r = {h_Content=c; h_HeapID=newID; h_Parent=id} in
              if insertHeapRow r then Some(true) else None
            else None
          else pushAt_int (id,v)
        else None
      else None

let push_int i =
  let root = getRoot() in
  let newID = freshHID () in
  match root with
  | [] → None
  | [rootRow] → match rootRow with
    | {h_Content = c ; h_HeapID = id; h_Parent = par ;} →
  if i > c then
    let r = {h_Content = i ; h_HeapID = newID; h_Parent = id ;} in
    if insertHeapRow r then Some(true) else None
  else
    if hasKeyHeap id then
      if hasKeyHeap par then
        if id = par then
    let rootRow' = {h_Content = i ; h_HeapID = id; h_Parent = par ;} in
    if updateHeapPK id rootRow' then
      let r = {h_Content = c ; h_HeapID = newID; h_Parent = id ;} in
      if insertHeapRow r then Some(true) else None
    else None
        else None
      else None
    else None
```

```
let rec rebalanceHeap id =
  let minM = getMinChild(id) in match minM with
  | [] → let res = deleteHeapPK id in res
  | [minRow] → match minRow with
    | {h_Content=mc; h_HeapID=mid; h_Parent=mpar} →
      if hasKeyHeap mid then
        let r = lookupHeapPK id in match r with
        | None → ()
        | (Some(u)) → match u with
          | {h_Content=rc ; h_HeapID=rid; h_Parent=rpar} →
            let v = {h_Content = mc; h_HeapID = id ; h_Parent = rpar} in
            updateHeapPK id v;
            let res = rebalanceHeap mid in res
      else ()

let pop_int () =
  let root = getRoot() in match root with
  | [] → None
  | [rootRow] → match rootRow with
    | {h_Content = c; h_HeapID = id; h_Parent = par} →
      (rebalanceHeap id; Some(c))
```

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
2. M. Benedikt, T. Griffin, and L. Libkin. Verifiable properties of database transactions. *Information and Computation*, 147(1):57–88, 1998.
3. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. In *Computer Security Foundations Symposium (CSF'08)*, pages 17–32. IEEE, 2008.
4. V. Benzaken and A. Doucet. Thémis: A database programming language handling integrity constraints. *VLDB Journal*, 4:493–517, 1995.
5. G. M. Bierman, A. D. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *International Conference on Functional Programming (ICFP)*, pages 105–116. ACM, 2010.
6. J. Borgström, A. D. Gordon, and R. Pucella. Roles, stacks, histories: A triple for Hoare. In *Journal of Functional Programming*, volume FirstView. Cambridge University Press, September 2010. An abridged version of this article was published in A. W. Roscoe, Cliff B. Jones, Kenneth R. Wood (eds.), *Reflections on the Work of C.A.R. Hoare*, Springer London Ltd, 2010.
7. M. A. Casanova and P. A. Bernstein. A formal system for reasoning about programs accessing a relational database. *ACM Transactions on Programming Languages and Systems*, 2(3):386–414, 1980.
8. A. J. Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *Programming Language Design and Implementation (PLDI)*, pages 122–133. ACM, 2010.
9. E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Formal Methods for Components and Objects (FMCO)*, volume 4709 of *LNCS*. Springer-Verlag, 2006.
10. A. Dasgupta, V. R. Narasayya, and M. Syamala. A static analysis framework for database applications. In *International Conference on Data Engineering (ICDE)*, pages 1403–1414. IEEE Computer, 2009.
11. J.-C. Filliâtre. Proof of imperative programs in type theory. In *Selected papers from the International Workshop on Types for Proofs and Programs (TYPES '98)*, 1657, pages 78–92. Springer, 1999.
12. C. Flanagan. Hybrid type checking. In *ACM Symposium on Principles of Programming Languages (POPL'06)*, pages 245–256, 2006.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
14. G. Gardarin and M. A. Melkanoff. Proving consistency of database transactions. In *Fifth International Conference on Very Large Data Bases*, pages 291–298. IEEE, 1979.
15. C. Gunter. *Semantics of programming languages*. MIT Press, 1992.
16. S. Krishnamurthi, P. W. Hopkins, J. Mccarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT scheme web server. *Journal of Higher-Order and Symbolic Computing (HOSC)*, 20(4):431–460, 2007.
17. J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Principles of Programming Languages (POPL)*, pages 237–248. ACM, 2010.
18. E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD Conference*, page 706. ACM, 2006.

19. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: dependent types for imperative programs. In *International Conference on Functional Programming (ICFP'08)*, pages 229–240. ACM, 2008.
20. M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, 2004.
21. S. Peyton Jones and P. Wadler. Comprehensive comprehensions. In *Haskell '07*, pages 61–72. ACM, 2007.
22. S. Ranise and C. Tinelli. *The SMT-LIB Standard: Version 1.2*, 2006.
23. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI'08)*, pages 159–169. ACM, 2008.
24. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3–4):289–360, 1993.
25. M. Serrano, E. Gallesio, and F. Loitsch. Hop: a language for programming the web 2.0. In *Object-oriented programming systems, languages, and applications (OOPSLA '06)*, pages 975–985. ACM, 2006.
26. T. Sheard and D. Stemple. Automatic verification of database transaction safety. *ACM Transactions on Database Systems*, 14(3):322–368, 1989.
27. N. Swamy, J. Chen, and R. Chugh. Enforcing stateful authorization and information flow policies in Fine. In *European Symposium on Programming Languages and Systems (ESOP)*, 2010.
28. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Apress, 2007.
29. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.
30. P. Wadler. Functional programming: An angry half-dozen. In *Advances in Database Programming Languages*, volume 1369 of *LNCS*, pages 25–34. Springer, 1997.
31. H. Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.