From Spreadsheets to Relational Databases and Back*

Jácome Cunha[1], João Saraiva[1], and Joost Visser[2]

[1] Departamento de Informática, Universidade do Minho, Portugal
[2] Software Improvement Group, The Netherlands

Abstract. This paper presents techniques and tools to transform spreadsheets into relational databases and back. A set of data refinement rules is introduced to map a tabular datatype into a relational database schema. Having expressed the transformation of the two data models as data refinements, we obtain for free the functions that migrate the data. We use well-known relational database techniques to optimize and query the data. Because data refinements define bidirectional transformations we can map such database back to an optimized spreadsheet.
We have implemented the data refinement rules and we have constructed tools to manipulate, optimize and refactor Excel-like spreadsheets.

1 Introduction

Spreadsheet tools can be viewed as programming environments for non-professional programmers. These so-called "end-user" programmers vastly outnumber professional programmers [28].

As a programming language, spreadsheets lack support for abstraction, testing, encapsulation, or structured programming. As a result, they are error-prone. In fact, numerous studies have shown that existing spreadsheets contain redundancy and errors at an alarmingly high rate [24, 27, 29, 30].

Spreadsheets are applications created by single end-users, without planning ahead of time for maintainability or scalability. Still, after their initial creation, many spreadsheets turn out to be used for storing and processing increasing amounts of data and supporting increasing numbers of users over long periods of time. To turn such spreadsheets into database-backed multi-user applications with high maintainability is not a smooth transition, but requires substantial time and effort.

In this paper, we develop techniques for smooth transitions between spreadsheets and relational databases. The basis of these techniques is the fundamental insight that spreadsheets and relational databases are formally connected by a data refinement relation. To find this relation we discover functional dependencies in spreadsheet data by data mining techniques. These functional dependencies can be exploited to derive a relational database schema. We then apply data

* This work is partially funded by the Portuguese Science Foundation (FCT) under grants SFRH/BD/30231/2006 and SFRH/BSAB/782/2008.

calculation laws to the derived schema in order to reconstruct a sequence of refinement steps that connects the relational database schema back to the tabular spreadsheet. Each refinement step at the schema level is witnessed by bidirectional conversion steps at the data level, allowing data to be converted from spreadsheet to database and vice versa. Our approach is to employ techniques for bidirectional transformation of types, values, functions, and constraints [32], based on data refinement theory [23].

We have implemented data refinement rules for converting between tabular and relational datatypes as a library in the functional programming language Haskell [25]. On this library, frontends were fitted for the exchange formats used by the spreadsheet systems Excel and Gnumeric. We have constructed two tools (a batch and an interactive version) to read, optimize, refactor and query spreadsheets. The tools get as argument a spreadsheet in the Excel or Gnumeric format and they have two different code generators: the SQL code generator, that produces SQL code to create and populate the corresponding relational database, and an Excel/Gnumeric code generator that produces a (transformed) spreadsheet.

This paper is organized as follows: Section 2 presents a motivating example that is used throughout the paper. Section 3 briefly discusses relational databases and functional dependencies. In Section 4 we define data refinements and framework for constraint-aware two-level transformation. In Section 5 we present the refinement rules to map databases into spreadsheets. In Section 6 we describe the libraries and tools constructed to transform and refactor spreadsheets. Section 7 discusses related work and Section 8 contains the conclusions. In Appendix we show the API of our library.

2 Motivating Example

Throughout the paper we will use a well-known example spreadsheet taken from [8] and reproduced in Figure 1. This sheet stores information about a housing renting system, gathering information about clients, owners and rents. It also stores prices and dates of renting. The name of each column gives a clear idea of the information it represents.

For the sake of argument, we extend this example with two additional columns, named *totalDays* (that computes the days of renting by subtracting the column

	A	B	C	D	E	F	G	H	I	J	K
1											
2	clientNo	propertyNo	cName	pAddress	rentStart	rentFinish	totalDays	rentPerDay	total rent	ownerNo	oName
3	cr76	pg4	john	6 Lawrence St.	1/7/00	8/31/01	602.00	50.00	30100.00	co40	tina
4	cr76	pg16	john	5 Novar Dr.	9/1/01	9/1/02	365.00	70.00	25550.00	co93	tony
5	cr56	pg4	aline	6 Lawrence St.	9/2/99	6/10/00	282.00	50.00	14100.00	co40	tina
6	cr56	pg36	aline	2 Manor Rd	10/10/00	12/1/01	417.00	60.00	25020.00	co93	tony
7	cr56	pg16	aline	5 Novar Dr.	11/1/02	8/10/03	282.00	70.00	19740.00	co93	tony

Fig. 1. A spreadsheet representing a property renting system.

rentFinish to *rentStart*) and *total rent* (that multiplies the total number of days of renting by the rent-per-day value, *rentPerDay*). As usual in spreadsheets, these columns are expressed by formulas.

This spreadsheet defines a valid model to represent the information of the renting system. However, it contains redundant information. For example, the displayed data specifies the house renting of two clients (and owners) only, but their names are included 5 times. This kind of redundancy makes the maintenance and update of the spreadsheet complex and error-prone. A mistake is easily made, for example by mistyping a name and thus corrupting the data.

The same information can be stored without redundancy. In fact, in the database community, techniques for database normalization are commonly used to minimize duplication of information and improve data integrity [31, 11]. Database normalization is based on the detecting and exploiting functional dependencies inherent in the data [5]. Can we leverage these database techniques for spreadsheets? Based on the data in our example spreadsheet, we would like to discover the following functional dependencies:

$$clientNo \rightharpoonup cName$$
$$ownerNo \rightharpoonup oName$$
$$propertyNo \rightharpoonup pAddress, rentPerDay, ownerNo, oName$$
$$clientNo, propertyNo \rightharpoonup rentStart, rentFinish, total\ rent, totalDays$$

We say that an attribute b (the *consequent*) is functionally dependent on attribute a (the *antecedent*), if a uniquely determines b (notation: $a \rightharpoonup b$). For instance, the client number functionally determines his/her name, since no two clients have the same number.

After discovering these dependencies we would like to infer a relational database schema which is optimized to eliminate data redundancy. This schema can then be used, either to store the data in a relational database management system, or to create an improved spreadsheet. Figure 2 presents such an optimized spreadsheet for our example. This new spreadsheet consists of four tables (bold boxes) and the redundancy present in the original spreadsheet has been eliminated. As expected, the names of the two clients (and owners) only occur once. As we will demonstrate in the remaining sections of this paper, the process of detecting functional dependencies, deriving a normalized database schema, and converting the data to the new format can be formalized and automated.

	A	B	C	D	E	F	G	H	I	J
1	clientNo	propertyNo	rentStart	rentFinish	total rent	totalDays	clientNo	cName	ownerNo	oName
2	cr76	pg4	1/7/00	8/31/01	30100	602	cr76	john	co40	tina
3	cr75	pg16	9/1/01	9/1/02	25550	365	cr56	aline	co93	tony
4	cr56	pg4	9/2/99	6/10/00	14100	282	propertyNo	pAddress	rentPerDay	ownerNo
5	cr56	pg36	10/10/00	12/1/01	25020	417	pg4	6 Lawrence St.	50	co40
6	cr56	pg16	11/1/02	8/10/03	19740	282	pg16	5 Novar Dr.	70	co93
7							pg36	2 Manor Rd	60	co93

Fig. 2. The spreadsheet after applying the third normal form refactoring.

After establishing a mapping between the original spreadsheet and a relational database schema, we may want to use SQL to query the spreadsheet. Regarding the house renting information, one may want to know who are the clients of the properties that where rented between January, 2000 and January 2002? Such queries are difficult to formulate in the spreadsheet environment. In SQL, the above question can be formulated as follows:

```
select clientNo from rent
    where rentStart between '1/01/00' and '1/01/02'
```

Below we will demonstrate that the automatically derived mapping can be exploited to fire such SQL queries at the original or the optimized spreadsheet.

In the next sections, we will formalize the correspondence between spreadsheets and relational schemas using data refinement rules. We will present formal proofs that guarantee their correctness. Moreover, we will present a framework that implements the transformation rules and includes frontends to well-known spreadsheet systems. In fact, the example presented in this section was processed by our framework.

3 From Functional Dependencies to RDB Schemas

This section explains how to extract functional dependencies from the spreadsheet data and how to construct the relational schema. We start by introducing some concepts related to relational databases. Then, we present an algorithm to extract functional dependencies from data. Finally, we describe how to use the FDs to create a relational schema.

Relational Databases and Functional Dependencies A *relational database* $DB = \{R_1, ..., R_n\}$ is a collection of named *relations* (or tables), $R_i \subseteq d_1 \times ... \times d_k$, defined over data sets not necessarily distinct. Each relation's R_i element $(d_1, ..., d_k)$ is called a *tuple* (or row) and each d_i is called an *attribute*. Each tuple is uniquely identified by a minimum nonempty set of attributes called *primary key* (PK). It could be the case of existing more then one set suitable for becoming the primary key. They are designated *candidate keys* and only one is chosen to becomes primary key. A *foreign key* (FK) is a set of attributes within one relation that matches the primary key of some relation. A *relational database schema* (RDB) is a set of relation schemas each of which being a set of attributes. These concepts are illustrated in Figure 3.

Another important concept is *functional dependency* (FD) between sets of attributes within a relation [5]. A set B is functionally dependent on a set A, denoted $A \rightharpoonup B$, if each value of A is associated with exactly one value of B.

The normalisation of a database is important to prevent data redundancy. Although, there are more normal forms, in general, a RDB is considered normalised if it respects the third normal form (3NF) [8], that is, if it respects the second normal formal (2NF) and all the non-key attributes are only dependent

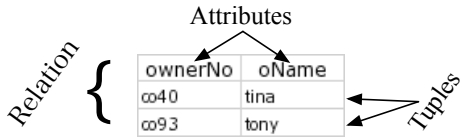


Fig. 3. An example of a relation that represents part of our example.

on the key attributes. A relation respects the 2NF if it is in the first normal form (1NF) and its non-key attributes are not functionally dependent on part of the key. Finally, the 1NF is respected if each element of each row contains only one element.

In order to define the RDB schema, we use the data mining algorithm FUN [18] to compute the FD given a spreadsheet, and then database techniques, namely Maier's algorithm [16], to compute the RDB schema in the 3NF.

We have expressed FUN as the HASKELL *fun* function. Next, we execute this function with our running example (the arguments *propSch* and *propRel* correspond to the first and remaining lines of the spreadsheet, respectively).

$*$ *ghci*⟩*fun propSch propRel*
ownerNo $\rightharpoonup$ *oName*
clientNo $\rightharpoonup$ *cName*
totalDays $\rightharpoonup$ *clientNo*, *cName*
propertyNo $\rightharpoonup$ *pAddress*, *rentPerDay*, *ownerNo*, *oName*
pAddress $\rightharpoonup$ *propertyNo*, *rentPerDay*, *ownerNo*, *oName*
...

The FDs derived by the FUN algorithm depend heavily on the quantity and quality of the data. Thus, for small samples of data, or data that exhibits too many or too few dependencies, the FUN algorithm may not produce the desired FDs. For instance, in our running example and considering only the data shown on Figure 1, the FUN algorithm does not induce the following FD *clientNo*, *propertyNo* $\rightharpoonup$ *rentStart*, *rentFinish*, *total rent*, *totalDays*.

3.1 Spreadsheet Formulas

Functional dependencies are the basis for defining the RDB schema. The FUN algorithm, however, may compute redundant FDs which may have a negative impact on the design of the RDB schema. In this section, we discuss characteristics of spreadsheets that can be used to define a more precise set of functional dependencies.

Spreadsheets use formulas to define the values of some elements in terms of other elements. For example, in the house renting spreadsheet, the column *totalDays* is computed by subtracting the column *rentFinish* to *rentStart*, and it is usually written as follows `G3 = F3 - E3`. This formula states that the

values of `F3` and `E3` determine the value of `G3`, thus inducing the following functional dependency: $rentStart, rentFinish \rightharpoonup totalDays$. Note also that *totalDays* is the primary key of a FD produced by the FUN algorithm, namely $totalDays \rightharpoonup clientNo, cName$. Primary keys, however, must be constants rather than formulas. Thus, such FDs should be eliminated.

Formulas can have references to other formulas. Consider, for example, the second formula of the running example `I3 = G3 * H3`, which defines the total rent by multiplying the number of days by the value of the rent. Because `G3` is defined by another formula, the values that determine `G3` also determine `I3`. As a result, the two formulas induce the following FDs:

$rentStart, rentFinish, rentPerDay \rightharpoonup total\ rent$
$rentStart, rentFinish \rightharpoonup totalDays$

Functional dependencies induced by formulas are added to the ones computed by the FUN algorithm. In genereal a spreadsheet formula of the form $X_0 = f(X_1, \ldots, X_n)$ induces the following functional dependency: $X_1, \ldots, X_n \rightharpoonup X_0$. In spreadsheet systems, formulas are usually introduced by copying them through all the elements in a column, thus making the FD explicit in all the elements. This may not always be the case and some elements can be defined otherwise (e.g. by using a constant value or a different formula). In this case, no functional dependency is induced.

3.2 Computing the RDB Schema

Having computed the functional dependencies, we can now construct the schema of the RDB. Maier in [16] defined an algorithm called `synthesize` that receives a set of FDs and returns a relational database schema respecting the 3NF.

begin synthesize :
Input a set of FDs F
Output a complete database schema for F
1. find a reduced, minimum annular cover G for F;
2. for each CFD $(X_1, X_2, ..., X_n) \rightharpoonup Y$ in G, construct a relational schema $R = X_1X_2...X_nY$ with designated keys $K = \{X_1, X_2, ..., X_n\}$;
3. return the set of relational schemas constructed in step 2.

end synthesize

This concise, but complex algorithm works as follows: To find a minimum annular cover G for F we start by compute a minimum cover G' for F. G' is minimum if it has as few FDs as any equivalent set of FDs. Transform G' into G is simple: just combine the FDs with equivalent left sides into *compound functional dependencies* (CFDs) having the form $(X_1, X_2, ..., X_n) \rightharpoonup Y$ where $X_1, ..., X_n, Y$ are sets of FDs and the left sets are equivalent.

Now we need to reduce the set of CFDs and this is achieved when all the CFDs into the set are reduced. A CFD is reduced if no left set contains any shiftable attributes and the right side contains no *extraneous* attributes. An attribute is

shiftable if it can be removed from the left side of the FD and inserted into the right side without changing the equivalence of the set of FDs. An attribute is *extraneous* if it can be removed from the FD without changing the equivalence of the set of FDs.

We have implemented this algorithm in HASKELL as the *synthesize* function. It gets as argument the functional dependencies (produced by the FUN) and returns a set of CFD. Next, we execute *synthesize* with the FDs induced by our running example (sets are represented by the predefined HASKELL lists).

$*\ ghci\rangle synthesize \circ fun\ propSch\ propRel$
$([ownerNo], [oName]) \rightharpoonup []$
$([clientNo], [cName]) \rightharpoonup []$
$([totalDays]) \rightharpoonup [cName]$
$([propertyNo], [pAddress], [rentPerDay]) \rightharpoonup [oName]$
$([rentStart, rentFinish, rentPerDay]) \rightharpoonup [total\ rent]$
$([rentStart, rentFinish]) \rightharpoonup [totalDays]$
...

Each CFD defines several candidate keys for each table. However, to fully characterise the RDB schema we need to chose the primary key from those candidates. To find such key we use a simple algorithm: first, we produce all the possible tables using each candidate key as a primary key. For example, the second CFD above expands to two possible tables with the same attributes: one has *clientNo* as primary key and in the other is *cName* the primary key. Next, we choose the table which has the smallest PK, since in general a 'good' table has the smallest key as possible. The more attributes the key determines the best. A final cleanup is necessary: we remove FD that all their attributes are already represented in other FDs. We also merge two FDs whenever antecedents/consequents are subsets. The final result is listed bellow.

$ownerNo \rightharpoonup oName$
$clientNo \rightharpoonup cName$
$propertyNo \rightharpoonup pAddress, rentPerDay, ownerNo$
$clientNo, propertyNo \rightharpoonup rentStart, rentFinish, total\ rent, totalDays$

As a final step, the set of foreign keys has to be computed by detecting which primary keys are referenced in the consequent of the FD.

Next, we show the the RDB schema derived for the house renting system example. The RDB is represented as a tuple of tables. A table is a map between the PK and the remaining attributes. This datatype is constrained by an invariant, defining the foreign keys, which will be explained in detail in the next section.

$$
\begin{array}{l}
(clientNo \times propertyNo \rightharpoonup rentStart \times rentFinish \times total\ rent \times totalDays \times \\
\quad clientNo \rightharpoonup cName \times \\
\quad\quad (propertyNo \rightharpoonup pAddress \times rentPerDay \times ownerNo \times \\
\quad\quad\quad ownerNo \rightharpoonup oName)_{inv1})_{inv2} \\
\textbf{where} \\
\quad inv1 = \pi_{ownerNo} \circ \rho \circ \pi_1 \subseteq \delta \circ \pi_2 \\
\quad inv2 = \pi_{clientNo} \circ \delta \circ \pi_1 \subseteq \delta \circ \pi_1 \circ \pi_2 \wedge \pi_{propertyNo} \circ \delta \circ \pi_1 \subseteq \delta \circ \pi_1 \circ \pi_2 \circ \pi_2
\end{array}
$$

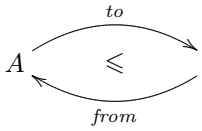
The tables are indexed by a binary function that represents the foreign keys restriction. This notation and operators are introduced in the next section.

4 Constraint-aware Rewriting

As we have explained before, the mapping between the spreadsheet and the RDB models is performed through data refinements using the 2LT system. Thus, before we present the data refinement rules to map spreadsheets into databases, let us briefly describe data refinements and the 2LT system[3].

4.1 Datatype Refinement

Data refinement theory provides an algebraic framework for calculating with datatypes. Refining a datatype A to a datatype B can be captured by the following diagram:



where $\begin{cases} to : A \to B \text{ is an injective and total relation;} \\ from : B \to A \text{ a surjective function;} \\ from \cdot to = id_A \text{ (identity function on } A); \end{cases}$

We will use $A \leqslant^{to}_{from} B$ as a short notation to the above diagram.

Refinements can be composed, that is, if $A \leqslant^{to}_{from} B$ and $B \leqslant^{to'}_{from'} C$ then $A \leqslant^{to' \cdot to}_{from \cdot from'} C$. Also, transformation in specific parts of a datatype must be propagated to the global datatype in which they are embedded, that is, if $A \leqslant^{to}_{from} B$ then $FA \leqslant^{Fto}_{Ffrom} FB$ where F is a functor that models the context of the transformation. A functor captures *i)* the embedding of local datatypes inside global datatypes and *ii)* the lifting of value-level functions to and $from$ on the local datatypes to value-level transformations on the global datatypes. In the particular case where the refinement works in both directions we have an isomorphism $A \cong B$.

A common example is that maps are the implementation for lists [9] – $A^\star \leqslant^{seq2index}_{list} N \rightharpoonup A$ – where *seq2index* creates a map (or finite function, here represented as Map) where the keys are the indexes of the elements of the list. $list$ just collects the elements in the map. For more details about data refinements the reader is referred to [17, 19, 23].

Consider now a RDB with two tables, $A \rightharpoonup B$ and $A \rightharpoonup C$. Suppose that the key of the first table is a foreign key to the key of the second one. This is represented with the *datatype constraint* $\delta \circ \pi_1 \subseteq \delta \circ \pi_2$, where π_1 and π_2 represent left and right projection, respectively, and δ denotes the domain of a map.

[3] 2LT stands for Two-Level Transformations. The tool is available at `http://code.google.com/p/2lt/`.

Constraints on data types are modelled as boolean functions which distinguishes between legal values and values that violate the constraint. A data type A with a constraint ϕ is represented as A_ϕ where $\phi : A \to Bool$ is a total function So, our example becomes as follows: $((A \rightharpoonup B) \times (A \rightharpoonup C))_{\delta \circ \pi_1 \subseteq \delta \circ \pi_2}$. Further details about constraint data types can be found in [4, 20].

4.2 Two-Level Transformations with Constraints

The data refinement theory presented in the previous section was implemented as a rewriting system named 2LT in Haskell [4, 9]. We will now briefly introduce this system.

A type-safe representation of types and functions is constructed using generalised algebraic datatypes (GADTs) [26]. To represent types, the following GADT is used:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Type}\ t\ \textbf{where} \\
\quad \mathit{String} :: \mathit{Type}\ \mathit{String} \\
\quad [\cdot] \quad :: \mathit{Type}\ a \to \mathit{Type}\ [a] \\
\quad \cdot \rightharpoonup \cdot \ :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ a \rightharpoonup b \\
\quad \cdot \times \cdot :: \mathit{Type}\ a \to \mathit{Type}\ b \to \mathit{Type}\ (a, b) \\
\quad \mathit{Maybe} :: \mathit{Type}\ a \to \mathit{Type}\ (\mathit{Maybe}\ a) \\
\quad \cdot_{\cdot} \quad :: \mathit{Type}\ a \to \mathit{PF}\ (a \to \mathit{Bool}) \to \mathit{Type}\ a \\
\quad \ldots
\end{array}
$$

Each refinement rule is encoded as a two-level rewriting rule:

$$
\begin{array}{l}
\textbf{type}\ \mathit{Rule} = \forall\ a\ .\ \mathit{Type}\ a \to \mathit{Maybe}\ (\mathit{View}\ (\mathit{Type}\ a)) \\
\textbf{data}\ \mathit{View}\ a\ \textbf{where}\ \mathit{View} :: \mathit{Rep}\ a\ b \to \mathit{Type}\ b \to \mathit{View}\ (\mathit{Type}\ a) \\
\textbf{data}\ \mathit{Rep}\ a\ b = \mathit{Rep}\{\, \mathit{to} = \mathit{PF}\ (a \to b), \mathit{from} = \mathit{PF}\ (b \to a)\,\}
\end{array}
$$

Although the refinement are from a type a to a type b, this can not be directed encoded since the type b is only known when the transformation completes, so the type b is represented as a *view* of the type a. A view means that given a function *to* which transforms a type a into a type b and a vice versa function *from* it is possible to construct b from a.

These functions are represented in a point-free style, that is, without any variables. Its representation is accomplished by the following GADT:

$$
\begin{array}{ll}
\textbf{data}\ \mathit{PF}\ a\ \textbf{where} & \\
\quad \pi_1 & :: \mathit{PF}\ ((a, b) \to a) \\
\quad \pi_2 & :: \mathit{PF}\ ((a, b) \to b) \\
\quad \mathit{list2set} & :: \mathit{PF}\ ([a] \to \mathit{Set}\ a) \\
\quad \rho & :: \mathit{PF}\ ((a \rightharpoonup b) \to \mathit{Set}\ b) \\
\quad \delta & :: \mathit{PF}\ ((a \rightharpoonup b) \to \mathit{Set}\ a) \\
\quad \mathit{CompList} & :: \mathit{PF}\ ([(a, b)] \to [(b, b)]) \\
\quad \mathit{ListId} & :: \mathit{PF}\ ([(a, b)] \to [(b, b)])
\end{array}
$$

$$
\begin{array}{ll}
\cdot^{\star} & :: PF\ (a \to b) \to PF\ ([a] \to [b]) \\
\cdot^{\star} & :: PF\ (a \to b) \to PF\ (Set\ a \to Set\ b) \\
\cdot \wedge \cdot & :: PF\ (Pred\ a) \to PF\ (Pred\ a) \to PF\ (Pred\ a) \\
\cdot \circ \cdot & :: PF\ (b \to c) \to PF\ (a \to b) \to PF\ (a \to c) \\
\cdot \triangle \cdot & :: PF\ (a \to b) \to PF\ (a \to c) \to PF\ (a \to (b, c)) \\
\cdot \times \cdot & :: PF\ (a \to b) \to PF\ (c \to d) \to PF\ ((a, c) \to (b, d)) \\
\cdot \subseteq \cdot & :: PF\ (a \to Set\ b) \to PF\ (a \to Set\ b) \to PF\ (Pred\ a) \\
Tables2table & :: PF\ ((a \rightharpoonup b, c \rightharpoonup d) \to (a \rightharpoonup b, Maybe\ d)) \\
Table2tables & :: PF\ ((a \rightharpoonup b, Maybe\ d) \to (a \rightharpoonup b, c \rightharpoonup d)) \\
Table2sstable & :: PF\ ((a \rightharpoonup b) \to [(a, b)]) \\
Sstable2table & :: PF\ ([(a, b)] \to (a \rightharpoonup b))
\end{array}
$$

...

To represent datatypes with constraints the following new *Type* constructor is used:

$\cdot.\ :: Type\ a \to PF\ (a \to Bool) \to Type\ a$

Its first argument is the type to constraint and the second one is the PF function representing the constraint.

Each refinement rule can only be applied to a specific datatype, for instance, a refinement on the type A can not be applied to the type $A \times B$. To allow this some rule-based combinators were created:

$$
\begin{array}{ll}
nop :: Rule & \text{-- identity} \\
\triangleright :: Rule \to Rule \to Rule & \text{-- sequential composition} \\
\oslash :: Rule \to Rule \to Rule & \text{-- left-biased choice} \\
many :: Rule \to Rule & \text{-- repetition} \\
once :: Rule \to Rule & \text{-- arbitrary depth rule application}
\end{array}
$$

Using combinators, rules can be combined in order to create a full rewrite systems.

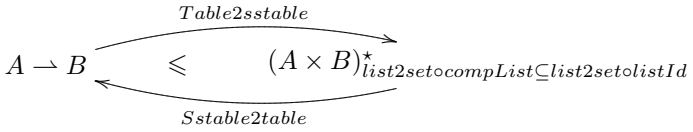
5 From a Database to a Spreadsheet: The Rules

In our approach spreadsheets are represented by a product of spreadsheet tables (from now designated as *sstables*). A *sstable* is a product of rows and each row is itself a product of values. Although, we wish to transform a spreadsheet into a relational database, we will introduce rules to map databases into spreadsheets since the former are a refinement of the later. Thus, we will use the RDB schema constructed, as explained in Section 3.2, and refine it to a spreadsheet. Because we do this data type migration within the 2LT system, we obtain for free the function that we will use to migrate the data of the spreadsheet into the database.

Next, we describe several date refinements needed to transform a relational database into a spreadsheet. We also present a strategy to apply these refinements in order to obtain the desirable result.

5.1 Refining a Table to a sstable

It is possible to transform a table (a map with key of type A and values of type B) into a sstable (a list of tuples) and vice-versa as long as there is a constraint imposing that there exists a FD between the elements in the column of type A and the column of type B:



Here, *list2set* transforms a list into a set, *compList* sees a list as a relation and compose it with its inverse. *listId* is the list resulting from transforming the *id* relation into a list. This definition of FD is based on the definition of FD presented in [21]. From now on this invariant will be designated *fd*. The proof of this data refinement can be found in [22].

The rule is implemented in HASKELL as follows.

```
table2sstable :: Rule
table2sstable (a ⇁ b)_i = return $ View rep [a × b]_inv'
  where
    inv' = trySimplify (i ∘ Sstable2table ∧ fd)
    rep = Rep{to = Table2sstable, from = Sstable2table}
```

where *trySimplify* is a rule that simplifies the invariant.

If the relational table has already an invariant, then it is composed with the invariant of the new type concerning with the FD. The resulting invariant is then simplified. This is just part of the rule's implementation since another function is necessary to encode the rule when the initial argument does not have an associated invariant.

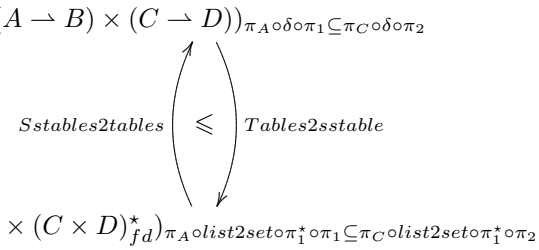
Let us use this rule to map the table with information about clients of our running example.

```
∗ ghci⟩table2sstable (clientNo ⇁ cName)
Just (View (Rep⟨to⟩⟨from⟩) [clientNo × cName]_fd)
```

The result of this rule is a datatype modelling a spreadsheet that constains the attribute of the database. The invariant *fd* guarantees that the functional dependency is now present in the datatype. Moreoever, the returned *to* and *from* functions are the migration functions needed to map the data between the two models.

5.2 Refining Tables with Foreign Keys on Primary Keys

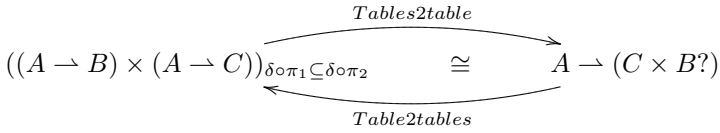
A pair of tables where the primary key of the first table is a foreign key to the primary key of the second table, can be refined to a pair of sstables using the following law:



The invariant guarantees that exists a FK from the PK of the first table to the PK of the second table. The π_A projection has type $\pi_A : A \to E$ and $\pi_C : C \to E$. A must be a tuple of the form $A_1 \times \ldots \times E \times \ldots \times A_n$ and C of the form $C_1 \times \ldots \times E \times \ldots \times C_n$. This allows that just part of the PK of each table is used in the FK definition. The proof of this refinement corresponds to apply twice the refinement presented in Section 5.1. The invariant must be updated to work on the new type. The HASKELL function *table2sstable* implements the rule.

```
table2sstable :: Rule
table2sstable ((a ⇁ b) × (c ⇁ d))_{π_A∘δ∘π_1 ⊆ π_C∘δ∘π_2} =
    return $ View rep ([a × b]_fd × [c × d]_fd)_inv
  where
    inv = π_A ∘ list2set ∘ π_1^⋆ ∘ π_1 ⊆ π_C ∘ list2set ∘ π_1^⋆ ∘ π_2
    rep = Rep{ to = Table2sstable × Table2sstable,
               from = Sstable2table × Sstable2table }
```

A particular instance of this refinement occurs when π_A is the identity function. In this case all the attributes of the PK of the first table, are FKs to part of the attributes of the PK of the second table. Another instance of this refinement is when $\pi_C = id$, that is, part of the attributes of the PK of the first table reference all the attributes of the PK of the second table. Both cases are very similar to the general one and they are not shown here. A final instance of this rule presents π_A and π_C has the identity function meaning that all the attributes of the PK of the first table are FKs to all the attributes of the PK of the second table. In this case the refinement changes as we show next.



The values of the second table have a ?[4] because it can be the case that some keys in the second table are not referenced in the first one. This happens because it is not always true that all the keys have a foreign usage. The proof of such an isomorphism is as follows.

[4] This symbol means optional. it is also representable as $A + 1$. In HASKELL it is represented by the datatype *Maybe* $x =$ *Just* $x \mid$ *Nothing*.

$$
\begin{array}{ll}
& A \rightharpoonup (C \times B?) \\
\cong & \quad \{\ A? \cong 1 \rightharpoonup A\} \qquad (1) \\
& A \rightharpoonup (C \times (1 \rightharpoonup B)) \\
\cong & \quad \{\ A \rightharpoonup (D \times (B \rightharpoonup C)) \cong ((A \rightharpoonup D) \times (A \times B \rightharpoonup C))_{\pi_1^\star \circ \delta \circ \pi_2 \subseteq \delta \circ \pi_1}\} \qquad (2) \\
& ((A \rightharpoonup C) \times (A \times 1 \rightharpoonup B))_{\pi_1^\star \circ \delta \circ \pi_2 \subseteq \delta \circ \pi_1} \\
\cong & \quad \{\ A \cong A \times 1\} \qquad (3) \\
& ((A \rightharpoonup C) \times (A \rightharpoonup B))_{\delta \circ \pi_2 \subseteq \delta \circ \pi_1}
\end{array}
$$

Proofs of rules 1, 2, and 3 can be found in [9], [4], and [23]. A function in HASKELL was created to define this isomorphism.

```
tables2table :: Rule
tables2table ((a ⇀ b) × (a ⇀ c))_{δ∘π1 ⊆ δ∘π2} =
    return $ View rep (a ⇀ c × Maybe b)
  where
    rep = Rep{ to = Tables2table, from = Table2tables }
```

Note that each of these rules has a dual one, that is, for each rule refining a pair $A \times B$ there exists another one refining the pair $B \times A$, with the appropriate invariant.

5.3 Refining Tables with Foreign Keys on Non Primary Keys

In the previous section we have introduced refinement rules to manipulate tables with FKs to PKs. In this section we present another set of rules to deal with FKs in the non-key attributes. The diagram of the general rule is presented below.

$$
\begin{array}{ccc}
((A \rightharpoonup B) \times (C \rightharpoonup D))_{\pi_B \circ \rho \circ \pi_1 \subseteq \pi_C \circ \delta \circ \pi_2} & & \\
\Big\uparrow Sstables2tables' & \leqslant & \Big\downarrow Tables2sstables' \\
((A \times B)^\star_{fd} \times (C \times D)^\star_{fd})_{\pi_B \circ list2set \circ \pi_2^\star \circ \pi_1 \subseteq \pi_C \circ list2set \circ \pi_1^\star \circ \pi_2} & &
\end{array}
$$

The proof of this refinement corresponds again to apply twice the refinement presented in Section 5.1. The invariant must be updated to work on the new type too. The function *tables2sstables* implements this refinement.

```
tables2sstables' ((a ⇀ b) × (c ⇀ d))_{πB∘ρ∘π1 ⊆ πC∘δ∘π2} =
    return $ View rep ([a × b]_fd × [c × d]_fd)_{πB∘list2set∘π2*∘π1 ⊆ πC∘list2set∘π1*∘π2}
```

where
$rep = Rep\{\, to = Table2sstable \times Table2sstable,$
$\qquad\qquad from = Sstable2table \times Sstable2table \,\}$

This refinement has three other particular cases. One where $\pi_B = id$, another where $\pi_C = id$, and finally when $\pi_B = \pi_C = id$. In these cases the refinement is the same, only the invariant changes. The proof and implementation of this rule are very similar to the general case and so not shown here.

Let us consider again our running example. In order to use this rule we consider two FDs where all the PK of second table is referenced by part of the non-key attributes of the first one.

$*\ ghci\rangle$**let** $prop = propertyNo \rightharpoonup pAddress \times rentPerDay \times ownerNo$
$*\ ghci\rangle$**let** $owner = ownerNo \rightharpoonup oName$
$*\ ghci\rangle maps2tables\ (prop \times owner)_{\pi_{ownerNo} \circ \rho \circ \pi_1 \subseteq \delta \circ \pi_2}$
$Just\ (View\ (Rep\langle to\rangle\langle from\rangle)$
$\qquad (prop' \times owner')_{\pi_{ownerNo} \circ list2set \circ \pi_1^\star \circ \pi_1 \subseteq list2set \circ \pi_1^\star \circ \pi_2})$
where
$prop' = [propertyNo \times pAddress \times rentPerDay \times ownerNo]$
$owner' = [ownerNo \times oName]$

In this case the two initial pairs are transformed into two lists. The constraint is updated to work with such structures.

5.4 Data Refinements as a Strategic Rewrite System

The individual refinement rules can be combined into a compound rules and full transformation systems using the strategy combinators shown in Section 4.2. In particular, we define a compound rule to map a RDB to a spreadsheet:

$rdb2ss :: Rule$
$rdb2ss = simplify \rhd$
$\quad (many\ ((aux\ tables2sstables) \oslash (aux\ tables2sstables'))) \rhd$
$\quad (many\ (aux\ table2sstable))$
where
$\quad aux\ r = ((once\ r) \rhd simplify) \rhd many\ ((once\ r) \rhd simplify)$

This function starts by simplifying the invariants. Then the *tables2sstables* rule (defined in Section 5.2) is applied exhaustively (with *aux*) to transform tables into sstables. After that the *tables2sstables′* rule (Section 5.3) is applied. In a final step, the remaining maps are transformed using the *table2sstable* (Section 5.1) rule. After each rule has been applied a simplification step is executed. This strategy requires the simplification of the invariants because pattern matching is performed not only on the type representations, but also on the invariants. The *simplify* rule is defined as follows:

$simplify :: Rule$
$simplify = many\ (prodsrdb \oslash mapsrdb \oslash others \oslash myRules) \rhd compr$

The functions *prodsrdb*, *mapsrdb*, *others*, and *myRules* are functions that simplify invariants. We include here the code of the first one. It simplifies products:

$$prod\_def :: Rule$$
$$prod\_def\ (Func\ (a \times b)\ \_)\ (f \times g) = success\ \texttt{"Prod-Def"}\ ((f \circ \pi_1) \triangle (g \circ \pi_2))$$
$$prod\_def\ \_\ \_ = fail\ \texttt{"Cannot apply Prod-Def!"}$$

6 The HaExcel Framework

HaExcel is a framework to manipulate, transform and query spreadsheets. We provide an overview of the various parts of the framework and we demonstrate its use by example.

6.1 The Framework

HaExcel is implemented in Haskell and consists of the following parts:

Library A generic/reusable library to map spreadsheets into relational database models and back: This library contains an algebraic data type to model a (generic) spreadsheet and functions to transform it into a relational model and vice versa. Such functions implement the refinement rules introduced in Section 5. The library includes two code generator functions: one that produces the SQL code to create and populate the database, and a function that generates Excel/Gnumeric code to map the database back into a spreadsheet. A MySQL database can also be created and manipulated using this library under HaskellDB [7, 15]. The API of HaExcel is included in Appendix A.

Front-ends A frontend to read spreadsheets in the Excel and Gnumeric formats: The frontend reads spreadsheets in the portable XML documents using the *UMinho Haskell Libraries* [1]. We reuse the spatial logic algorithms from the UCheck project [3] to discover the tables stored in the spreadsheet. The first row of each table is used as labels and the remaining elements are assumed to be data.

Tools Two spreadsheet tools: A batch and a online tool that allow the users to read, transform, refactor and query spreadsheets.

6.2 The House Renting Spreadsheet Revisited

In Section 2 we have informally presented the refactoring of the house renting spreadsheet. In this section, we will use the HaExcel library functions to perform such refactoring automatically.

The library function *ss2rdb* (see Appendix A) reads the Excel spreadsheet from a file and produces the RDB schema and the migrated data. This function performs the following steps: First, it uses the function *readSS* to read the spreadsheet stored in the file received as argument. This function not only reads the

Excel file, but it also detects the tables in the spreadsheet (using UCheck functions). Next, the *rel2rdb* function uses the already presented *fun* and *synthesize* functions to compute the functional dependencies and construct the relational database schema. The data refinement rules are used to map the spreadsheet into a database model. The *ss2rdb* returns a pair containing the the database schema and the migrated data, that conforms to the returned schema. The migrated data is computed using the functions automatically derived during the refinement. Next, we show the returned RDB schema obtained for the running example.

$$
\begin{array}{l}
(\mathit{clientNo} \times \mathit{propertyNo} \rightharpoonup \mathit{rentStart} \times \mathit{rentFinish} \times \mathit{total\ rent} \times \mathit{totalDays} \times \\
\quad \mathit{clientNo} \rightharpoonup \mathit{cName} \times \\
\quad\quad (\mathit{propertyNo} \rightharpoonup \mathit{pAddress} \times \mathit{rentPerDay} \times \mathit{ownerNo} \times \\
\quad\quad\quad \mathit{ownerNo} \rightharpoonup \mathit{oName})_{\mathit{inv1}})_{\mathit{inv2}} \\
\textbf{where} \\
\quad \mathit{inv1} = \pi_{\mathit{ownerNo}} \circ \rho \circ \pi_1 \subseteq \delta \circ \pi_2 \\
\quad \mathit{inv2} = \pi_{\mathit{clientNo}} \circ \delta \circ \pi_1 \subseteq \delta \circ \pi_1 \circ \pi_2 \wedge \pi_{\mathit{propertyNo}} \circ \delta \circ \pi_1 \subseteq \delta \circ \pi_1 \circ \pi_2 \circ \pi_2
\end{array}
$$

and, the migrated data.

$$
\begin{array}{l}
\{((\mathit{cr76}, \mathit{pg4}), (1/7/00, (8/31/01, (30100, 602)))), \\
\ ((\mathit{cr75}, \mathit{pg16}), (9/1/01, (9/1/02, (25550, 365)))), \\
\ ((\mathit{cr56}, \mathit{pg4}), (9/2/99, (6/10/00, (14100, 282)))), \\
\ ((\mathit{cr56}, \mathit{pg36}), (10/10/00, (12/1/01, (25550, 417)))), \\
\ ((\mathit{cr56}, \mathit{pg16}), (11/1/02, (8/10/03, (19740, 282))))\} \\
\quad (\{(\mathit{cr76}, \mathit{john}), (\mathit{cr56}, \mathit{aline})\}, \\
\quad\quad (\{(\mathit{pg4}, (6\ \mathit{Lawrence\ St}\ ., (50, \mathit{co40}))), (\mathit{pg16}, (5\ \mathit{Novar\ Dr}\ ., (70, \mathit{co93}))), \\
\quad\quad\quad (\mathit{pg36}, (2\ \mathit{Manor\ Rd}, (60, \mathit{co93})))\}, \\
\quad\quad\quad\quad \{(\mathit{co40}, \mathit{tina}), (\mathit{co93}, \mathit{tony})\}))
\end{array}
$$

The returned RDB schema defines a 3NF database, consisting of four tables. This corresponds exactly to the four tables in the spreadsheet of Figure 2. The same happens with the data: it corresponds to the data stored in that spreadsheet. Having computed the relational database tables and the migrated data, we can now export it into SQL and back into refactored spreadsheet.

The SQL Backend: This backend generates SQL code which creates the database according to the derived RDB schema. This is basically a simple SQL `create` instruction based on the RDB schema. Furthermore, it produces SQL code to insert the migrated data in the database, and, again, this corresponds to a SQL `insert` instruction with the migrated data as argument. Because some values of the spreadsheet are defined through formulas, we generate also a SQL trigger that models the spreadsheet formulas which are used to update the database and guarantee its integrity. Next, we present the trigger induced by the two formulas of our running example:

```
create trigger ssformulas before insert on tbl
  for each row begin
```

$set\_\ new\ .\ totalDays = new\ .\ rentFinish - new\ .\ rentStart;$
$set\_\ new\ .\ total\_rent = new\ .\ rentPerDay * new\ .\ totalDays;$
$end;$

The SQL code to create the database and insert the migrated data can be obtained from the HaExcel webpage. We omit them here.

The Spreadsheet Backend: It generates the spreadsheet in the XML-based Excel/Gnumeric format. The data refinements define bidirectional transformations, so we can map the optimized relational database back into a spreadsheet. Thus, this backend implements a spreadsheet refactoring tool. Figure 2 shows the result of using this backend for the house renting system. Because, the spreadsheet formulas are stored in the database via triggers, the refactored spreadsheet contains such formulas too.

7 Related Work

Frost *et al* [14] describe a method of providing a spreadsheet-like interface to a relational database. The method starts from meta-data about the data organization in the spreadsheet. Using this meta-data a bidirectional connection is established between the spreadsheet and the database. No intermediate storage or transformation is performed on the data, and the database schema is a direct reflection of the spreadsheet organization captured by the metadata. No specific method is given for providing the required meta-data in an automated fashion. By contrast, our approach includes automated discovery of spreadsheet organization, derivation of a normalized database schema, and calculational construction of the bi-directional connection between database and spreadsheet, which may include complex transformations of the data that is exchanged.

Pierce *et al* [6, 13] have addressed the *view-update problem* for databases with combinators for bi-directional programming, called *lenses*. A database view constructed by composition of lenses allows updates at the view level to be pushed back to the database level. Lenses are similar, but fundamentally different from data refinements (a formal comparison is given by Oliveira [23]). Lenses can be used to fit a flattened, spreadsheet-like view onto a normalized relational database. Though our approach starts from a spreadsheet and derives a relational database, part of our solution involves the calculation of bi-directional conversion functions, starting from the database schema and working back to the initial spreadsheet. It would be interesting to investigate whether lenses can be calculated similarly.

The 2LT project [32] uses two-level strategic term rewriting for coupled transformation of data schemas, instances [9], queries [10], and constraints [4], in particular, hierarchical-relational data mappings. We adopted the same techniques and demonstrated their application to transformation from non-structured data (spreadsheets) to relational databases. We added new transformation rules and strategies with corresponding proofs. We integrated the rewriting machinery with

algorithms for detection of tables and functional dependencies and generation of relational schemas.

Erwig *et al* have applied to spreadsheets various software engineering and programming language principles, including type inference [3], debugging [2], and object-oriented design [12]. In particular, the UCheck system [3] detects errors in spreadsheets through automatic detection of headers and (numeric) unit information. We have adopted some of the spatial logic algorithms from UCheck in order to detect table boundaries.

8 Conclusions

In this paper, we have explored the relation between spreadsheets and relational databases. In particular, we have made the following contributions.

1. We have extended the 2LT framework for two-level transformations with constraint-aware
2. conversion rules between relational and tabular data structures. The correctness of these rules is supported with proofs.
3. We have shown how these rules can be combined into a strategic rewrite
4. system that transforms a relational schema into a tabular schema and on the fly derives conversion functions between these schemas.
5. We have combined this rewrite system with methods for discovering tables in spreadsheets, deriving functional dependencies from the data contained in them, and deriving relational schemas from these functional dependencies.
6. To these algorithmic components, we have connected importers and exporters for SQL and spreadsheet formats.
7. We have shown how the resulting system can be employed to convert spreadsheets to relational databases and back. This allows refactoring of spreadsheets to reduce data redundancy, migration of spreadsheet applications to database applications, and advanced querying of spreadsheets with SQL queries.

Notwithstanding these contributions, our approach presents a number of limitations that we hope to remove in future. For example, we are currently supporting only set of commonly used formulas, which remains to be enlarged to a wider range.

More importantly, the algorithm for inferring functional dependencies needs improvement. This algorithm does not work well for small sets of data and is sensitive to "accidental" patterns in the data. Some well-chosen heuristics and limited user interaction could alleviate this problem.

In the two-level rewrite system, syntactic matching is performed on representations of constraints. Such syntactic matching could be generalized to verification of logical implication of the actual constraint and the required constraint.

With respect to formulas and queries, we are not yet exploiting some interesting opportunities. For example, a formula or query expressed in terms of the source spreadsheet may be composed with the backward conversion function to

obtain a query on the target database or refactored spreadsheet. Such migrations of queries have been explored in [10].

Finally, we are currently ignoring the possibility of having repeated rows in the source spreadsheet. Also the order of rows is not taken into account. Thus, we are effectively treating spreadsheet tables as sets of rows rather than bags or lists of rows. The data refinement theory that underpins our approach can be exploited to support these alternative perspectives where order and repetition are considered relevant.

Availability The HaExcel library and tools are available from the homepage of the first author.

References

1. UMinho Haskell Software – Libraries and Tools - http://wiki.di.uminho.pt/twiki/bin/view/research/pure/puresoftware.
2. R. Abraham and M. Erwig. GoalDebug: A spreadsheet debugger for end users. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 251–260. IEEE Computer Society, 2007.
3. R. Abraham and M. Erwig. UCheck: A spreadsheet type checker for end users. *J. Vis. Lang. Comput.*, 18(1):71–95, 2007.
4. T.L. Alves, P.F. Silva, and J. Visser. Constraint-aware Schema Transformation. In *The Ninth International Workshop on Rule-Based Programming*, 2008.
5. C. Beeri, R. Fagin, and J.H. Howard. A complete axiomatization for functional and multivalued dependencies in database relations. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 47–61, 1977.
6. A. Bohannon, J.A. Vaughan, and B.C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*, 2006.
7. B. Bringert, A. Höckersten, C. Andersson, M. Andersson, M. Bergman, V. Blomqvist, and T. Martin. Student paper: HaskellDB improved. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 108–115, New York, NY, USA, 2004. ACM.
8. T. Connolly and C. Begg. *Database Systems, A Practical Approach to Design, Implementation, and Management.* Addison-Wesley, 3 edition, 2002.
9. A. Cunha, J.N. Oliveira, and J. Visser. Type-safe Two-level Data Transformation. In J. Misra et al., editors, *Proc. Formal Methods, 14th Int. Symp. Formal Methods Europe*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.
10. A. Cunha and J. Visser. Strongly typed rewriting for coupled software transformation. *ENTCS*, 174(1):17–34, 2007. Proc. 7th Int. Workshop on Rule-Based Programming (RULE 2006).
11. C. J. Date. *An Introduction to Database Systems, 6th Edition.* Addison-Wesley, 1995.
12. G. Engels and M. Erwig. ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*, pages 124–133. ACM, 2005.

13. J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, May 2007.
14. B.H. Frost and S.D. Stanton. Spreadsheet-based relational database interface, January 2008. US Patent 20080016041.
15. D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain Specific Languages (DSL'99)*, pages 109–122, Austin, Texas, October 1999.
16. D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1983.
17. C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.
18. N. Novelli and R. Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, pages 189–203, London, UK, 2001. Springer-Verlag.
19. J.N. Oliveira. A reification calculus for model-oriented software specification. *Formal Asp. Comput.*, 2(1):1–23, 1990.
20. J.N. Oliveira. "Fractal" Types: an Attempt to Generalize Hash Table Calculation. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, June 1998.
21. J.N. Oliveira. Functional dependency theory made 'simpler'. Technical Report PURe-05.01.01, DI-Research, January 2005.
22. J.N. Oliveira. Pointfree foundations for (generic) lossless decomposition. 2007. Submitted.
23. J.N. Oliveira. Transforming data by calculation. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, volume 5235 of *Lecture Notes in Computer Science*. Springer, To appear, 2008.
24. R.R. Panko. Spreadsheet errors: What we know. what we think we can do. *Proceedings of the Spreadsheet Risk Symposium, European Spreadsheet Risks Interest Group (EuSpRIG)*, July 2000.
25. S. Peyton Jones. Haskell 98: Language and libraries. *J. Funct. Program.*, 13(1):1–255, 2003.
26. S. Peyton Jones, G. Washburn, and S. Weirich. Wobbly types: type inference for generalised algebraic data types. Technical Report MS-CIS-05-26, Univ. of Pennsylvania, July 2004.
27. K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality control in spreadsheets: A software engineering-based approach to spreadsheet development. In *HICSS '00: Proc. 33rd Hawaii International Conference on System Sciences-Volume 4*, page 4006, Washington, DC, USA, 2000. IEEE Computer Society.
28. C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 207–214, 20-24 Sept. 2005.
29. T. SH Teo and M. Tan. Quantitative and qualitative errors in spreadsheet development. *Proc. Thirtieth Hawaii Int. Conf. on System Sciences*, 3:149–156, 1997.
30. M. Tukiainen. Uncovering effects of programming paradigms: Errors in two spreadsheet systems. *12th Workshop of the Psychology of Programming Interest Group (PPIG)*, pages 247–266, April 2000.
31. J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I*. Computer Science Press, 1988.

32. J. Visser. Coupled transformation of schemas, documents, queries, and constraints. *Electr. Notes Theor. Comput. Sci.*, 200(3):3–23, 2008.

A API

```
  -- type definitions
type R a = Attributes a
type Relation a = [[a]]
data Attributes a = Atts{ atts :: [Attribute a] }
data Attribute a = Att{ att :: a }
type FD a = (Attributes a, Attributes a)
type FDS a = [FD a]
type CFD a = (LeftSets a, RightSets a)
type CFDS a = [CFD a]
type LeftSets a = [Attributes a]
type RightSets a = Attributes a
type LKS a = [LK a]
data LK a = LK{ candidate :: Candidate a, cardinality :: Int,
                quasiClosure :: Attributes a, closure :: Attributes a }
type Candidate a = Attributes a
data Box where Box :: Type a → a → Box
  -- functions to manipulate FDs
fun :: Monad m ⇒ R b → Relation a → m (LKs b)      -- implementation of Fun
lks2fds :: Eq a ⇒ LKS a → FDS a                    -- transforms LKS into FDS
rel2rdb :: R a → Relation b → Box                  -- produces a RDB schema
synthesize :: FDS a → CFDS a                       -- implementation of synthesize
  -- import/export functions
readSS :: FilePath → IO [Sheet Fml]                -- reads a SS file; returns tables
ss2boxes :: FilePath → IO [(Maybe Box, Maybe Box)] -- returns type tables
ss2rdb :: FilePath → IO [Maybe Box]                -- SS to RDB type
ss2SQL :: FilePath → FilePath → IO ()              -- SS to an SQL script
genSQL :: [(Maybe Box, Maybe Box)] → [[String]]    -- produces SQL queries
ss2DB :: FilePath → MySQLOptions → IO ()           -- SS to a MySQL DB
genDB :: [(Maybe Box, Maybe Box)] → MySQLOptions → IO ()  -- gives a DB
ss2SS :: FilePath → FilePath → IO ()               -- SS to a new SS
genSS :: [Maybe Box] → Gmr' Workbook → Gmr' Workbook      -- produces a SS
  -- refinement law functions
table2sstable :: Rule                              -- rule from Section 5.1
tables2table Rule                                  -- rule from Section 5.2
tables2sstables :: Rule                            -- rule from Section 5.3
rdb2ss :: Rule                                     -- strategy: rule from Section 5.4
```