Formal Methods

XML to SQL conversion

using Type-safe Two-level Data Transformation

Diogo Paulo da Fonte Lapa (38164)
Flávio Miguel Xavier Ferreira (38187)
Hugo José Pereira Pacheco (38204)

Departamento de Informática da Universidade do Minho
Campus de Gualtar - Braga - Portugal

July 25, 2006

Abstract

Data migration is the transferring of data between storage types, formats, or computer systems [**?**], and represents a core procedure in modern information systems. Data migration can be achieved through creation of data mappings between data types, which imply complex transformations over types. Transformations in data types can be designed as a two-level data transformation, which consist of a type-level transformation of a data format coupled with value-level transformations of data instances corresponding to that format.
We provide a backend for fully automated data migration between *XML* and *SQL* , based on a formalized type-safe two-level data transformation, written in *Haskell* .
Keywords: SQL, XML, XML Schema, Two-level transformation, Type-safe, Data type, Data mappings.

Contents

Chapter 1

Introduction

The Extensible Markup Language (*XML*) is a W3C-recommended general-purpose markup language for creating special-purpose markup languages, capable of describing many different kinds of data. In other words, XML is a way of describing data. A *XML* file can contain the data too, as in a database.[**?**] *SQL* is the most popular computer language used to create, modify, retrieve and manipulate data from relational database management systems[**?**], and therefore, is also a way of representing data. Despite both technologies describe a way in how to represent data, they follow very different programming paradigms: Data types in *XML* are mostly described as a composition of more basic types. A *XML* schema defines a type of *XML* document in terms of constraints upon what elements and attributes may appear, along with the definition of types and their relations to each other. *SQL* , as a database system, represents data as a product of tables, where a table is represented as map between products of nillable or not elements.

Therefore, migration between both types can be described in data mappings, defined as transformations on the data types and corresponding data instances. This coupled type and values transformation is called a *Two-level Transformation* . We will consider a general type-safe *Two-level Transformation* technique formally implemented in *Haskell* [**?**]. The transformations are performed in a fully automated translation process over the entire data structure and corresponding data instances.

In this project, an interface for a *Two-level Transformation* from *XML* to *SQL* has been developed, as two separate data mappings: the first from *XML* to the type-safe representation; and the second from the transformed *Type* representation to *SQL* .

In Chapter 2 we explain the details of the transformation processes for both *XML* and *SQL* , for which we rely on various tools and techniques.

Chapter 2

The conversion process

Although this project is aimed to migrate *XML* data types into *SQL* tables, the platform which enables translations between the original language representation and a type-safe representation is not oriented, in the sense that the conversion is not only one way, but can occur from the language to *Type* and reverse.

The final goal would be to allow the whole reverse conversion, from *SQL* to *XML* , currently not supported by the *Two-level Transformation* engine.

Each translation is implemented as the composition of two or more data mappings.

All the conversions can be classified in two main classes: *XML* conversions or *SQL* conversions.

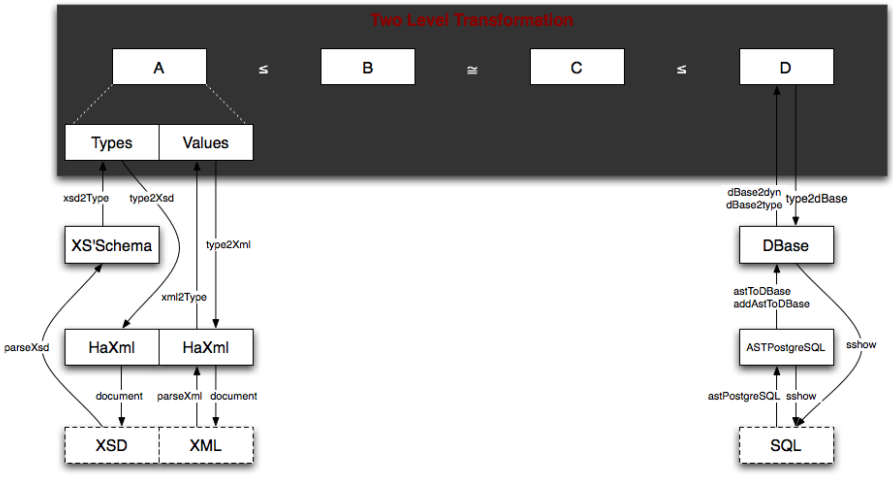

Figure 2.1: Detailed diagram on *XML* to *SQL Two-level Transformation*

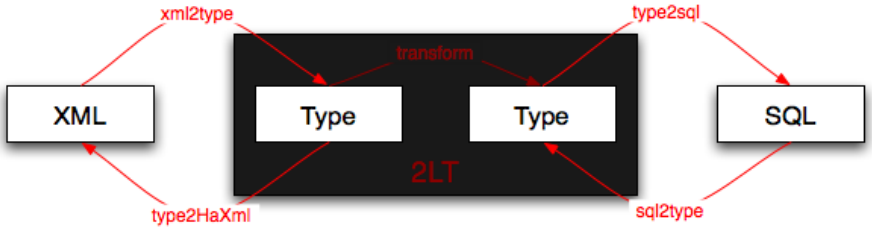

Figure 2.2: *XML* to *SQL Two-level Transformation* in general

2.1 Two-level Transformation

A *two-level data transformation* consists of a type transformation of a data format coupled with value-level transformations of data instances corresponding to that format. Examples of two-level data transformations include *XML* schema evolution coupled with document migration, and data mappings used for interoperability and persistence.

The *Two-level Transformation* engined used provides a formal treatment of two-level data transformations that is type-safe, in the sense that the well-formedness of the value-level transformations with respect to the type-level transformation is guarded by a strong *Type* system. The transformation relies on various techniques for generic functional programming to operationalize the formalization in *Haskell* .

The formalization addresses various *Two-level Transformation* scenarios, covering fully automated as well as user-driven transformations, and allowing transformations that are information-preserving or not. In each case, two-level transformations are disciplined by one-step transformation rules and type-level transformations induce value-level transformations. We demonstrate an example, hierarchical-relational mapping and subsequent migration of relational data induced by hierarchical format evolution.

In this project, the *Two-level Transformation* engine is used to convert a *XML* data type representation (a tree of nested products and co-products) into a *SQL* database table representation (a product of maps), both described over the generic type-safe *Haskell* data *Type* .

2.2 XML conversions

A data type can be represented by a couple of a type-level declaration and an instance of that type in the value-level. A standard *XML* file contains only a value-level layer, where the type is implicitly declared in the *XML* tree structure. This type representation can't distinguish between Strings or numeric types per example, what reduces the application of such structure.

Helped by this fact, a *XML Schema Definition* has been developed in order to extend *XML* to strongly-defined data types. Our understanding of the *XML Schema Definition* is based on the *World Wide Web Consortium* *XSD* specifications[**?**] and the *W3 Schools* XML schema tutorials[**?**]. When translating from input *XML* and *XSD* files to *Type* , the input files need to be parsed into

Haskell type-safe representations, isomorphic to the files language definition.

The tool used for parsing *XML* files is *HaXml* and Joost Visser's XSD parser for *XSD* files.

2.2.1 XML to Type

The conversion from *XML* to *Type* implies two major operations: initially create the *Type* representation of the *XML* schema; followed by a parsing of the corresponding values into the created type. The most sensible operation is in creating the *XML* schema representation, due to all the decisions in the schema analysis: *XSD* features a very extensive grammar, supporting very complex types. The same *Type* abstraction may have multiple *XSD* representations, increasing the ambiguity of the schema and, consequently, the sensitivity of the decision algorithm.

Consider the follow example, where from this *XSD* file,

```
<xs:element name="purchaseOrder1" type="myType1"/>
  <xs:element name="purchaseOrder2" type="myType2"/>
  <xs:element name="comment" type="xs:string"/>

  <xs:complexType name="myType1">
    <xs:sequence>
      <xs:element name="shipTo" type="xs:string"/>
      <xs:element name="billTo" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="orderDate" type="xs:date"/>
  </xs:complexType>

  <xs:complexType name="myType2">
    <xs:sequence>
      <xs:element name="name"   type="xs:string"/>
      <xs:element name="street" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
```

and a matching *XML* file,

```
<comment>Hello World!</comment>
```

a correspondig *Type* and values are generated.

```
Type Definition:
(Either (Tag "purchaseOrder1" (Prod (Prod (Tag "shipTo" String
    ) (Tag "billTo" String)) (Tag "orderDate" String))) (
    Either (Tag "purchaseOrder2" (Prod (Tag "name" String) (
    Tag "street" String))) (Tag "comment" String)))

Values:
Right (Right (comment="Hello World!"))
```

2.2.2 Type to XML

This may be the simplest step of all conversions. From each possible type, the *type2Xml* function maps it to an equivalent XSD representation, and binds it

to the value-level instance. Both the XSD and XML resulting instances are created in the *HaXml XML* type representation, which can be printed as valid XML. The interesting method in this process is the mapping from the *Two-level Transformation Type* to a *XML* schema type, informally defined as follows:

- the Item and Bool basic types are converted to *xs:integer*;
- the Char and String basic types are converted to *xs:string*;
- a list is represented with an element with attribute *maxOccurs* set to unbounded;
- sets are represented as lists;
- the maybe type (an either with the one of the branches as One) is represented as a an element with attribute *minOccurs* with value 0;
- an either is represented with a *xs:choice* element;
- products are represented as a sequence with two elements. Nested products are collapsed into a single sequence.

If applying the reverse conversion we get the following schema:

```
<?xml version='1.0' encoding='utf-8' ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.di.uminho.pt"
           xmlns="http://www.di.uminho.pt" elementFormDefault=
    "qualified"
  ><xs:element name="either1"
    ><xs:complexType
      ><xs:choice
        ><xs:element name="purchaseOrder1"
          ><xs:complexType
            ><xs:sequence
              ><xs:element name="shipTo" type="xs:string"
              /><xs:element name="billTo" type="xs:string"
              /><xs:element name="orderDate"
                            type="xs:string"/></xs:sequence></
    xs:complexType></xs:element
        ><xs:element name="purchaseOrder2"
          ><xs:complexType
            ><xs:sequence
              ><xs:element name="name" type="xs:string"
              /><xs:element name="street"
                            type="xs:string"/></xs:sequence></
    xs:complexType></xs:element
        ><xs:element name="comment"
                     type="xs:string"/></xs:choice></
    xs:complexType></xs:element></xs:schema>
```

and the *XML* file:

```
<?xml version='1.0' encoding='utf-8' ?>
<either1
  ><comment
    >Hello World!</comment></either1>
```

We can conclude that, even with the merge of nested products and co-products, the output *XSD* type is more obsolete than the original one, due to the extra elements and types. This could be improved with specific type generation for different sequences of types.

2.3 SQL Conversions

In a *SQL* relational database,a type is defined as a product of tables, and the values are the entries for each tables. In order to convert *SQL* statements into a *Type* representation, the intermediate type DBase is used, a database non-type-safe representation, where each table is splitted into primary key variables and normal variables, both for the type-level and value-level declarations.

2.3.1 Type to SQL

When converting the One *Type* to *SQL* , a assumption has to be made, as long as One doesn't have a *SQL* basic type equivalent. The assumption we've made is to create a *SQL* Boolean with name *TLT_One*, with value *true*.

The *Type* to *SQL* conversion may create or simply not remove inappropriate types, since relational database support only maps. For this reason, a safe version of *type2dBase* has been implemented, namely *type2dBaseSafe*, which removes all non supported types. Per example, if the *Type* $(A \rightharpoonup B) \times C$ would to be converted, the filter function would ignore the C *Type* and convert only the *Type* $A \rightharpoonup B$.

As an example, if we consider the *Type* :

```
Type Definition:
(Prod (Tag "Workers" (Map (Tag "id" Int) (Tag "name" String)))
     (Prod (Tag "Boss" (Map One (Tag "worker" Int))) (Tag "
    Salary" (Map (Tag "worker" Int) (Tag "amount" Int)))))

Values:
(Workers={id=1 ↦ name="John",id=2 ↦ name="Jack",id=3 ↦ name="
    Anna"},(Boss={() ↦ worker=1},Salary={worker=1 ↦ amount
    =1000,worker=2 ↦ amount=500,worker=3 ↦ amount=650}))
```

and convert it to *SQL* , we get the following output:

```
CREATE TABLE Workers
        (
         id              INTEGER NOT NULL
        ,name            VARCHAR(4) NOT NULL
        ,PRIMARY KEY (id)
        ) ;
INSERT INTO Workers VALUES (1,'John');
INSERT INTO Workers VALUES (2,'Jack');
INSERT INTO Workers VALUES (3,'Anna');

CREATE TABLE Boss
        (
```

```
        TLT_One                 BOOLEAN NOT NULL DEFAULT (True
    )
        ,worker          INTEGER NOT NULL
        ,PRIMARY KEY (TLT_One)
        ) ;
INSERT INTO Boss VALUES (True,1);

CREATE TABLE Salary
        (
         worker          INTEGER NOT NULL
        ,amount          INTEGER NOT NULL
        ,PRIMARY KEY (worker)
        ) ;
INSERT INTO Salary VALUES (1,1000);
INSERT INTO Salary VALUES (2,500);
INSERT INTO Salary VALUES (3,650);
```

2.3.2 SQL to Type

In the *SQL* to *Type* conversion, the *SQL* statements need to be converted to a valid database representation, what brings the need to an *Haskell SQL* parser. A *SQL* parser was developed as a separate project[**?**], and parses multiple *SQL* files into a single database non-safe representation defined by us (defined in the *DBase Haskell* type). The first step in this conversion is to create a type definition matching the tables structure, and then bind the table values to that type.

Recovering the previous example, if we apply convert the result *SQL* back to *Type* we get the initial *Type* :

```
Type Definition:
(Prod (Tag "Workers" (Map (Tag "id" Int) (Tag "name" String)))
     (Prod (Tag "Boss" (Map One (Tag "worker" Int))) (Tag "
    Salary" (Map (Tag "worker" Int) (Tag "amount" Int)))))

Values:
(Workers={id=1 ↦ name="John",id=2 ↦ name="Jack",id=3 ↦ name="
    Anna"},(Boss={() ↦ worker=1},Salary={worker=1 ↦ amount
    =1000,worker=2 ↦ amount=500,worker=3 ↦ amount=650}))
```

Chapter 3

Interface

The built interface for this project has several options in or order to allow iteration over the different steps of conversion. The existent options are provided as additional parameters, and are described in the application's help:

```
XML to SQL Two Level Transformation
Usage: xml2sql [options]
Options:
        --help                          Display this
    information
        --xml xmlfile xsdfile           Converts a XML
    file and its corresponding schema into valid SQL
    sentences (default)
        --xmltype xmlfile xsdfile       Converts a XML
    file and its corresponding schema into a data
    structure type representation
        --xmltransform xmlfile xsdfile  Converts a XML
    file and its corresponding schema into a database type
     representation
        --xmldbase xmlfile xsdfile      Converts a XML
    file and its corresponding schema into a database non-
    type-safe representation
        --parsexml xmlfile              Parses an input
    xml file into the HaXml data type
        --parsexsd xsdfile              Parses an input
    XML schema file into Joost Visser's XSD Abstract
    Syntax Tree
        --xsdtype xsdfile           Parser an input XML
    schema file into a type-safe representation
        --sql [sqlfiles]                Converts multiple
     SQL files into a database type-safe representation
        --sqldbase [sqlfiles]           Converts multiple
     SQL files into a database non-type-safe
    representation
```

More SQL parsing options are provided in the Happy SQL Parser project[**?**].

Chapter 4

Future Work

At the end of this project, we have achieved a minimum functionality, but considering that the implemented translations are not formally specified (in contrast with the *Two-level Transformation*) and bugs or unexpected behaviors safeness is not guaranteed.

A lot of effort still needs to be forwarded to the improvement of currently supported*XML* and *SQL* features:

- the *XSD* attributes *minOccurs* and *maxOccurs*, for values other than default (value 1), generate lists of elements. This approach is clearly not the best since, when reading *XML* values, we can't verify the exact number of elements, accepting lists of elements of any size. This problem would be easily overwhelmed by representing a a list of minimum length nested products, and the following elements as products of (Either One A) Considering we are trying to represent lists of elements A with 3 to 5 elements:

  $(A \times (A \times (A \times (One + (A \times (One + A))))))$

  This solution solves the problem of lists with n elements, but creates a large type, and therefore isn't the better approach. A better approach is to create a new type in *Type* , defining lists with minimum and maximum length.

- both *XSD* and *SQL* support basic types with numeric precision, such as double, and float types. For such types, as long as not precision type is defined in *Type* , we convert them to Int, and the precision values are ignored.

- one of the most complex *XSD* types is the xs:any, which represents any of the types declared in the whole schema or imported schemas. Currently, the xs:any is represented in the form of recursion over the functor of the schema root, what is equivalent to an either of all the types in the schema root. However, this representation is incorrect, because it should represent an either of all the defined types, and not only the ones in the schema's root. Although this representation is easy to implement, it carries an heavy load in the final type.

  We have not yet discovered a good solution for this issue.

- *XML Schema Definition* supports import and include of other schema files, along with the ability to redefine previously defined types in such imported/included schemas.

  The *XSD* redefine element redefines simple and complex types, groups, and attribute groups from an external schema.

  The *XSD* import element is used to add multiple schemas with different target namespace to a document.

  The *XSD* include element is used to add multiple schemas with the same target namespace to a document.

  Currently, all these features are parsed as redefine elements, as long as the other schema's definitions are merged with the actual schema's definitions, giving priority to the second group of definitions.

Similarly, due to the extensiveness of the languages, some *XML* and *SQL* features are not yet supported by our tool:

- mixed content in *XSD* complex types is not considered to be relevant and is not yet supported. This special elements allows a complex type to contain attributes, elements, and text.

- in *SQL* , the only supported constraints are the primary key constraints and the null/not null constraint. The primary key constraints is implicit in the keys of the Map *Type* , and the nillable constraint is represented by a Either A One *Type* . All the other constraints, such as the foreign key constraint, are lost in the conversion process, and would need the intermediate type to support constraints representation, what could be achieved by representing constraints in the form of pre-conditions, functions that would test the correctness of the tables. Even though, this is a very complex subject;

- restrictions in *XML* schema files are not supported by the *Type* representation, and are ignored.

- the notation element describes the format of non-XML data within an XML document. this advanced feature is completely not supported, since it has no influence in the final *XSD* data type.

We represent recursive types with the explicit fixpoint operator μ, as defined in [**?**]. If we look at the *XML* schema with simple recursion, defined in "src/examples/xml/xml12.xsd":

```
<xs:element name="A">
<xs:complexType>
  <xs:choice>
    <xs:element name="c" type="C" />
    <xs:element name="b" type="B" />
    <xs:element name="a" type="A" />
  </xs:choice>
</xs:complexType>
</xs:element>

<xs:complexType name="B">
```

```
    <xs:choice>
      <xs:element name="s" type="xs:string"/>
      <xs:element name="b" type="B"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="C">
    <xs:sequence>
      <xs:element name="s" type="xs:string"/>
      <xs:element name="i" type="xs:integer"/>
    </xs:sequence>
  </xs:complexType>
```

the corresponding type in *Haskell* is:

```
(Tag "A" (μ (((K (Tag "s" String)) :*: (K (Tag "i" Int))) :+: ((K
    (Tag "b" (μ ((K (Tag "s" String)) :+: ID)))) :+: ID))))
```

This method can't deal with multiple recursion, since when collapsing a recursive type reference in an already recursive type, the algorithm doesn't know which functor to apply for that recursive reference. For example, it we consider the abstract type A:

$$A = B + String$$
$$B = A + B$$

we can identify A and B as recursive types, what makes it impossible to substitute the reference to type B in the type A functor. At this report's moment, recursion is only supported at the type-level. The value-level binding for recursion is currently being developed.

As stated before, the implemented translations were not formally implemented and, therefore, we can not define any relation property between *XML* , *SQL* and *Type* . This derives that, once a conversion is made, the back conversion does not guarantee that the result will match the origin. An improvement would be to guarantee properties over the implemented translations, by proving the requirements of the representation and abstraction relations.

A *XML* schema is independent from the *XML* value-level, and consequently the same schema can be bound to multiple *XML* instances, representing values. This tool will allow the user to dump this multiple *XML* instances to *SQL* , and merge them together in a single database. This is possible because the schema type definition is similar in all files, and then the corresponding *SQL* tables will have the same format. However, *XML* schemas are often created for a specific *XML* file, and the migration of that data to *SQL* shall represent the best possible that value instance, this means, the type for the value instance can be optimized for the specific values it contains. Per example, if a type from the schema would be representable as (Either A B), but in the values-level only the left side (A) is used, we can ignore the right side and simplify the type to only the left branch. This method, provided as an extension, can be extended to the whole type and corresponding values, and help removing unwanted details in the type-level.

Due to problems encountered in the parser, the current *SQL* parser[**?**] supports only a minimum *SQL* scheme, validating only simple INSERT and CREATE statements. Another future work item would be to extend the parser to support the full *PostgreSQL* grammar.

Another possible future work would be to create a conversion for *XML* value-level only, considering that a *XML* schema would may not be available. Such conversion would create only string types, organized according to the *XML* tree implicit in the file declaration.

With the achievement of this project, we strongly believe that it helps enhancing the concept of automated data migration, as a crucial part of nowadays data storage systems.

Appendix A

XML to SQL conversion without recursion

A.1 "src/examples/xml/xml1.xml

```
<?xml version="1.0" encoding="utf-8"?>
 <Employees>
    <Employee ID="38164">
      <FirstName>Diogo</FirstName>
      <LastName>Lapa</LastName>
      <contact>111-111-1111</contact>
      <contact>dtlapa@yahoo.com</contact>
      <contact>AAA</contact>
    </Employee>
    <Employee2>Ze</Employee2>
    <Employee ID="38204">
      <FirstName>Hugo</FirstName>
      <LastName>Pacheco</LastName>
      <contact>666-666-1111</contact>
      <contact>hpacheco@gmail.com</contact>
    </Employee>
    <Employee ID="38187">
      <FirstName>Flavio</FirstName>
      <LastName>Ferreira</LastName>
      <contact>flavioxavier@gmail.com</contact>
    </Employee>
</Employees>
```

A.2 "src/examples/xml/xml1.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
                        targetNamespace="http://www.books.org"
                        xmlns="http://www.books.org"
                        elementFormDefault="qualified">

   <xs:element name="Employees">
```

```
        <xs:complexType>
        <xs:choice minOccurs="1" maxOccurs="unbounded">
           <xs:element name="Employee" type="EmployeeType"/>
           <xs:element name="Employee2" type="xs:string"/>
        </xs:choice>
        </xs:complexType>
    </xs:element>

        <xs:complexType name="EmployeeType">
           <xs:sequence>
              <xs:element name="FirstName" type="xs:string"
                          minOccurs="1" maxOccurs="1"/>
              <xs:element name="LastName" type="xs:string"
                          minOccurs="1" maxOccurs="1"/>
              <xs:element name="contact"type="xs:string"
                          minOccurs="1" maxOccurs="unbounded"/>
           </xs:sequence>
           <xs:attribute name="ID" form="unqualified" type="
     xs:string"/>
        </xs:complexType>
</xs:schema>
```

A.3 The data Type

```
Type Definition:
(Tag "Employees" (List (Either (Tag "Employee" (Prod (Prod (
    Tag "FirstName" String) (Prod (Tag "LastName" String) (
    List (Tag "contact" String)))) (Tag "ID" String))) (Tag "
    Employee2" String))))

Values:
Employees=[Left (Employee=((FirstName="Diogo",(LastName="Lapa"
    ,[contact="111-111-1111",contact="dtlapa@yahoo.com",
    contact="AAA"]))),ID="38164")),Right (Employee2="Ze"),Left
    (Employee=((FirstName="Hugo",(LastName="Pacheco",[contact=
    "666-666-1111",contact="hpacheco@gmail.com"]))),ID="38204")
    ),Left (Employee=((FirstName="Flavio",(LastName="Ferreira"
    ,[contact="flavioxavier@gmail.com"]))),ID="38187"))]
```

A.4 The transformed data Type

```
Type Definition:
(Prod (Prod (Map Int (Prod (Prod String String) String)) (Map
    (Prod Int Int) String)) (Map Int String))

Values:
(({0 ↦ (("Diogo","Lapa"),"38164"),2 ↦ (("Hugo","Pacheco"),"
    38204"),3 ↦ (("Flavio","Ferreira"),"38187")},{(0,0) ↦ "
    111-111-1111",(0,1) ↦ "dtlapa@yahoo.com",(0,2) ↦ "AAA"
    ,(2,0) ↦ "666-666-1111",(2,1) ↦ "hpacheco@gmail.com",(3,0)
    ↦ "flavioxavier@gmail.com"}),{1 ↦ "Ze"})
```

A.5 The SQL output

```
CREATE TABLE Table1
        (
         Column1                INTEGER NOT NULL
        ,Column2                VARCHAR(8) NOT NULL
        ,Column3                VARCHAR(8) NOT NULL
        ,Column4                VARCHAR(8) NOT NULL
        ,PRIMARY KEY (Column1)
        ) ;

INSERT INTO Table1 VALUES (0,'Diogo','Lapa','38164');

INSERT INTO Table1 VALUES (2,'Hugo','Pacheco','38204');

INSERT INTO Table1 VALUES (3,'Flavio','Ferreira','38187');


CREATE TABLE Table2
        (
         Column5                INTEGER NOT NULL
        ,Column6                INTEGER NOT NULL
        ,Column7                VARCHAR(22) NOT NULL
        ,PRIMARY KEY (Column5,Column6)
        ) ;

INSERT INTO Table2 VALUES (0,0,'111-111-1111');

INSERT INTO Table2 VALUES (0,1,'dtlapa@yahoo.com');

INSERT INTO Table2 VALUES (0,2,'AAA');

INSERT INTO Table2 VALUES (2,0,'666-666-1111');

INSERT INTO Table2 VALUES (2,1,'hpacheco@gmail.com');

INSERT INTO Table2 VALUES (3,0,'flavioxavier@gmail.com');


CREATE TABLE Table3
        (
         Column8                INTEGER NOT NULL
        ,Column9                VARCHAR(2) NOT NULL
        ,PRIMARY KEY (Column8)
        ) ;

INSERT INTO Table3 VALUES (1,'Ze');
```

Bibliography

[COV06] Alcino Cunha, José Nuno Oliveira, and Joost Visser.
Type-safe two level transformation.
Technical report, Departamento de Informática da Universidade do Minho, Campus de Gualtar, 4710-057 Braga, Portugal, 2006.

[LFP06] Diogo Lapa, Flávio Ferreira, and Hugo Pacheco.
Happy sql parser.
Technical report, Departamento de Informática da Universidade do Minho, 2006.

[Sch06] W3 Schools.
Introduction to xml schema, 2006.
http://www.w3schools.com/schema/schema_intro.asp.

[W3C06] W3C.
Xml schema, 2006.
http://www.w3.org/XML/Schema#dev.

[Wik06a] Wikipedia.
Data migration, 2006.
http://en.wikipedia.org/wiki/Data_migration.

[Wik06b] Wikipedia.
Sql, 2006.
http://en.wikipedia.org/wiki/Sql.

[Wik06c] Wikipedia.
Xml, 2006.
http://en.wikipedia.org/wiki/Xml.