Formal Methods

Graphical User Interface

for two-level transformation

Diogo Paulo da Fonte Lapa (38164)

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

March 2, 2007

Abstract

A two-level data transformation (2LT) consists of a type-level transformation of a data format coupled with value-level transformations of data instances corresponding to that format. A typical example of a 2LT consists in transforming XML schemas into relational database schemas, coupled with the migration of the XML documents to relational tables.

I have implemented a Graphical User Interface for 2LT. The core of the system consists of a type-safe Haskell library. In this paper I show how to use this system to import a data format, apply transformations on it and finally perform conversion on data instances based on these modifications.

Keywords Haskell, GUI, Gtk2Hs, 2LT, Transformation

Contents

1 Introduction

Coupled software transformation involves the modification of multiple software artifacts such that they remain consistent with each other. Two-level data transformation is a particular instance of coupled transformation, where the coupled artifacts are a data format on the one hand, and the data instances that conform to that format on the other hand.

In this project I will develop a **Graphical User Interface** (GUI) for *Two-level Transformation*. A GUI is a type of user interface using graphics for interaction with a computer which employs graphical images, widgets, along with text to represent the information and actions available to a user. The actions are usually performed through direct manipulation of the graphical elements. The key of this project is to implement an application which allow the user to perform *Two-level Transformation* through a user-friendly interface.

The core system is a *Haskell* library [COV06] already developed. It 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. It also provides suites of rule combinators as well as basic rules for format evolution. Using these basic rules and the rule combinators, we can compose sophisticated strategies for two-level transformation.

2 Problem formulation

A data type can be represented by a couple of a type-level declaration and an instance of that type in the value-level. Changes in data types call for corresponding changes in data values. Format evolution and data mappings are instances of what we call two-level transformations, where a type-level transformation (of the data type) determines or constrains value-level transformations (of the data instances).

I have to develop a tool which performs this kind of transformations on data types using a graphical interface. The interface should allow:

- Import of a data format, specified e.g. in *XML* Schema, *SQL*, *Haskell*
- Specification of rules to apply to the format, by selecting from a palette of rules and rule combinators, or by writing a high-level script
- Running the rules
- Inspecting the result and the derivation steps
- Apply the resulting data conversion function to data instances, either in backward or forward fashion

3 Solution and Algorithm Presentation

3.1 GUI Library

The first step it to decide which GUI Library to use. I was advised to use *wxHaskel*, built on top of *wxWidgets*. Due to problems related to compatibility with some versions of *GHC*, on previous experiences, I decided to use another one. After a small research about GUIs for *Haskell*, *Gtk2Hs* turned out to be the best solution for me. To point out some of the advantages of *Gtk2Hs* over other *Haskell* GUI libraries, I give a brief overview of what *Gtk2Hs* does better than most of the other libraries:

- Glade support
- Great API reference documentation
- Unicode support
- Memory management

3.1.1 Gtk2Hs

As I mentioned above, *Gtk2Hs* is a **Graphical User Interface** library for *Haskell*[Gtk]. It is based on *Gtk+*, a multi-platform toolkit for creating graphical user interfaces.

When talking about a GUI library, it is often useful to first explain some of the used terminology:

- **Widget:** a component of a graphical user interface that the user interacts with. Examples: button, label, scroll bar, ...
- **Container:** a widget which is able to contain other widgets
- **Box:** a container which aligns all of its widgets either in a vertical or horizontal way

Gtk2Hs is a powerful, user-friendly *Haskell* GUI library. The current API available is very useful already, and since *Gtk2Hs* hasn't reached 1.0 yet, it will only improve.

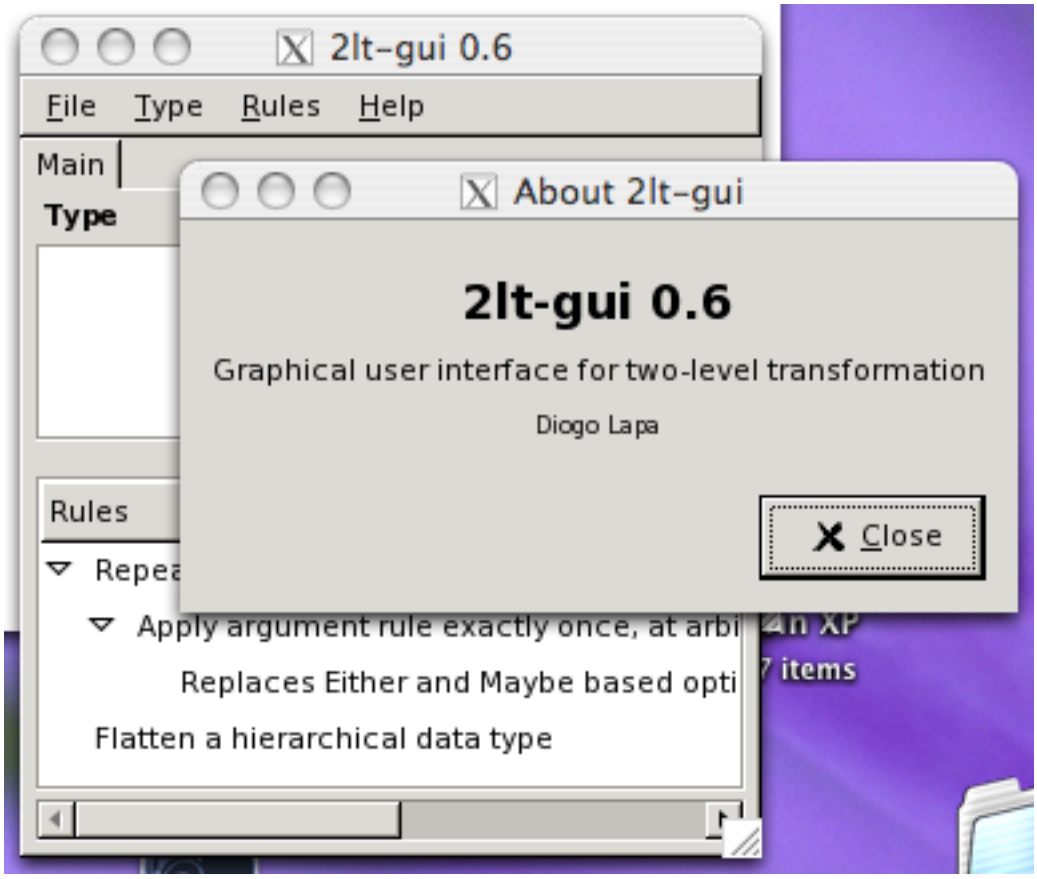


Figure 1: Screenshot from the GUI

3.1.2 Glade

Glade is a tool to create *XML* descriptions of a **Graphical User Interface**. Using Glade, it is not necessary to write application code to construct GUIs. Moreover, it lets the user redesign the GUI appearance without having to

touch the code. Glade allows *Gtk2Hs* to read the GUI definition at runtime. When the user wants to change some small things about the GUI, or even completely re-design it, no recompiling of the *Haskell* code is needed. Just make sure the new `*.glade` file has the same name as the last one, and contains the same widgets (with the same names) as the last GUI definition.

The use of Glade to create a GUI allows the developer of the application to concentrate on the interaction part of the application, instead of making sure it looks good.

3.2 Internal Data

The first stage of this GUI is to import a data type definition. It's possible to import from a *XML* Schema, *SQL* or a *Haskell* definition. To import *XML* and *SQL* I used front ends already implemented [LFP06]. The *Haskell* front end is not a real *Haskell* source parser. Instead of read full *Haskell* sources and get all the data types declared, this front end only reads strings representing *Type* structures in *Haskell* notation.

The imported type is parsed to a generic type defined by the *Two-level Transformation* library. This data type *Type*, is an example of a *generalized algebraic data type* (GADT), a recent *Haskell* extension that allows to assign more precise types to data constructors by restricting the variables of the data type in the constructor's result types.

In the next stage we define the transformation steps that we want to apply in the imported format. These steps are called rules. The approach is compositional, in the sense that full transformations are composed from basic transformation rules and rule combinators. The user can choose the rules from a interactive menu or by loading a high-level script.

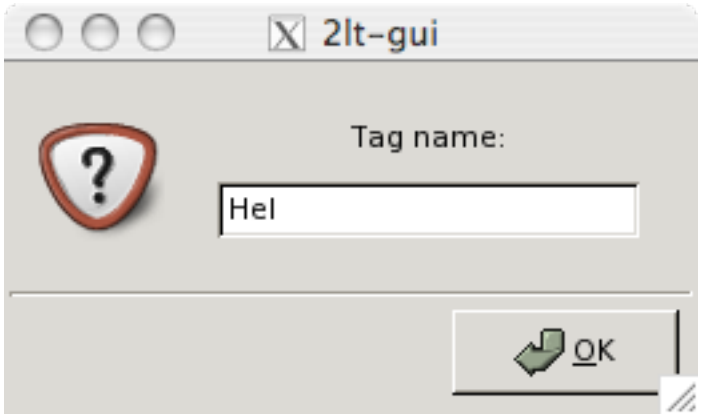


Figure 2: Example of popup window requesting information for a rule

The rules are saved in a *TreeView* widget. Each leaf node of the tree stores

a basic rule or an empty rule (node waiting for a rule). The combinators are stored in the non-leaf nodes. When you select a node of the tree, the next rule will be added in that position.

3.2.1 Front Ends

In order to embed the general transformation kernel into a language-specific transformation framework, there are some front ends for the relational database language *SQL*, the *Haskell* source language and the document markup language *XML*. These front ends specify functions for parsing and printing of types and values. Against the interface of the *FrontEnd* class, I can program an overloaded function that reads a file from the disc and parses to an internal type representation, the import function. It is also used to create another function to parse data instances, the last stage.

FrontEnd *Value-level*	*Type-level*
XML Document	Schema
SQL DML	DDL
Haskell String	String

Table 1: Front ends available

3.2.2 Rules Script

A rules script is a file with a high-level script defining a set of rules. To load this script into the application I used *Parsec*. *Parsec* is a free monadic parser combinator library for *Haskell*. It is simple, safe, well documented, has extensive libraries and good error messages.

This parser is very simple. The parser reads the file transforming the script in the *TreeView* widget's data structure. As explained before, it is just a tree with rules stored in the nodes. These rules are identified by a data type defined as follows.

```
1 data RulesNode = SeqComp | LeftBiasedChoice | Nop | Many
      | Once | All | Everywhere | AddField String String |
      AddAlt String | AllowRep | Inside String | AtTagNode
      String | Flatten | NormalizeTag | EitherToMaybe |
    ReplaceDefaultTags | Mapsum | EmptyNode deriving Eq
```

The syntax of the scripts is very similar to the *Haskell* source. However there are some differences:

- To represent strings it's possible to use either " or ' to encapsulate them.
- Nested strings are represented like in some script languages, you have to change between one notation or another (See script 2 in the examples).
- Rules receiving *Type* as argument, represent it as a string.

This is a small example how a script looks like:

```
1 script 1:
2         once nop >>> add_alt "One"
3
4 script 2:
5         inside "Format" (add_alt "Tag␣'Hello'␣String")
```

3.3 How to use

Two-level data transformation can be formalized in terms of data refinement theory, and can be modeled in *Haskell* as systems of type-changing rewrite rules. These rewrite rules operate on *Haskell* types. A data type A can be refined to a data type B if there is an injective, total function *to* (the representation function) and a surjective, possibly partial function *from* (the abstraction function) such that $from\ .\ to = id_A$, where id_A is the identity function on data type A. The inequations of data refinement theory can be used as rewrite rules that replace one data type by another. When applied left-to-right, will preserve or enrich information content, while applied right-to-left it will preserve or restrict information content.

This GUI is supposed to be user-friendly. I tried to create a layout for better comprehension of the process. I can divide the transformation process in three steps:

- Import data format and choose rules to apply
- Apply rules and inspect results on type level transformations
- Apply the resulting conversion to data instances and export to data formats

These three steps are performed in three different tabs in the *Notebook* widget.

3.3.1 Main Tab

The first one is the main tab. In this tab the user can see a *TextView* where the imported type is shown as a generic data type. There are three different ways to import data types available in the type section, inside the menu.

The other widget is the *TreeView* that contains the rules to apply as explained before. The user can add rules to the tree with the menu, in the rules section.

With a data type imported and some rules selected, it's possible to go to the next step, clicking the *Apply Rules* button. This will create a new tab, View.

Figure 3: Type menu

3.3.2 View Tab

Type and *View* are *generalized algebraic data types* (GADTs), an extension to the *Haskell* type system that allows (partially) instantiated type parameters in the result type of data constructors. The *Rule* type expresses that a two-level transformation step is a partial function that takes a type into a view of that type. Here we use a value-level representation of data types, where a value of *Type a* is the representation of type *a*. The *View* constructor expresses that a type a can be transformed into a type *b*, if there are functions *to* and *from*, bundled in the *Rep* constructor, that allow data conversion between *a* and *b*. Note that only the source type *a* escapes from the *View* constructor, while the target type *b* remains encapsulated — it is implicitly existentially quantified.

The View tab represents such a structure. You can see a *TextView* widget representing the source type that can be type-changed into the target type, and vice-versa, using the forward and backward functions. To use this two

functions, we also need data instances of the given types. The user has access to three ways of loading data instances (*XML*, *SQL*, *Haskell* source), using again the *FrontEnd* class.

Basically, in this tab the user has two data types, represented in a generic way, the source *Type a* and the target *Type b*. Its possible to populate each one with the forward and backward buttons and apply the transformation. The forward buttons parse a type value of type a and after the transformation the user gets a type $d$ata value according to the target type definition. The backward button is exactly the same but in the other direction. After parsing the value to a type b the transformation creates an instance of a type a value.

3.3.3 Result Tab

These data instances are stored in the last tab, the result tab. This is just a tab showing a *TextView* widget with the resulting data value printed.

Finally, the user can export the result data value and the corresponding type to a file in *XML*, *SQL* and *Haskell* formats. Again, I used the existing front ends to implement this.

4 Experiments

To exemplify the usage of the application I used an example of data mapping between *XML* and *SQL*. The required data mapping involves a format transformation from an *XML* schema to an *SQL* schema. The source schema (A.1) is imported. This will produce an internal data type representation of this schema (A.3). This representation is generic, no matter what kind of import we do, we always produce a data type of this *Type*.

This data type can be transformed. This is the next step, choose and apply the rules. My application has the ability to insert rules in two ways: choosing from a palette or loading them from a high-level script. I'll use the second option for this example. First I need to write an script in any text editor. Using basic rules and the rule combinators, I can compose sophisticated strategies for *Two-level Transformation*. For example, a hierarchical-relational mapping can be defined along the following script.

```
1 normalizeTag >>> (many (once eitherToMaybe)) >>> flatten
     >>> (many (once mapsum)) >>> (many (once
    replaceDefaultTags))
```

All the rules in this script are defined in the *Two-level Transformation* Library. It is not possible to create new rules, only combinations of existing rules.

The next step is to import the script: $Menu \rightarrow Rules \rightarrow Import\ from\ script$. This will parse the file and insert the respective nodes in the *TreeView* widget.

Since we have the rules and the data type (B.1), we can apply the rules: $Menu \rightarrow Type \rightarrow Apply\ Rules$. This action creates the View tab (B.2). This tab shows the target type (A.4) and offers the possibility to apply data conversion functions to data instances. I'll apply it to a *XML* instance of the source schema (A.2). This will create an instance of the target type in a new tab, Result tab (B.3).

After this, it is supposed to have the possibility to export the result couple of type-level and value-level. This feature is not yet implement. If I had implemented this, in this case, we could save the result to a SQL file, containing `CREATE` and `INSERT` statements.

This is an example of *Two-level Transformation*. There are others applications of this technic such as *format evolution*.

5 Future Work and Conclusion

There some features that are not working yet. Some due to lack of time, others to possible bugs, either from my application or from the *Two-level Transformation* library.

5.1 Intermediate Steps

The original idea was to show the intermediate steps when the user runs the given rules. The current version only show the first and the final step (source and target types). One possible implementation of this feature is to modify the *State Monad* used in the rules evaluation in a way that we can store the intermediate steps. Each evaluated rule adds an intermediate step to the state.

5.2 Import SQL files

When the user imports the data type from SQL files it's only possible to load a single SQL file. This is an issue because normally the data type is represented in multiple files instead of a single file. This is easily solved finding a way to choose various files to import. Maybe it's possible to modify the load file dialog such that the user can select more than one file, or create some intermediate structure to load each file at a time and import the merged result.

5.3 Backward Transformation and Export

The backward type-changing transformation and the export data types are not yet implemented.

5.4 Known Bugs

There are some known bugs that are not resolved yet:

- **Show a Dynamic type:** in the last stage, when the application shows the final result, there is some problem with the function used to print a Dynamic type (type and value). I am using a temporary *show* function based on a old version.
- **Error messages not caught:** some error messages are not being caught yet. This results in a fatal error instead of a popup window showing the message.

I still have some work to do but, at this stage, I have already a functional tool to perform *Two-level Transformation*. This **Graphical User Interface**simplifies the process of changing from one data type to another, giving the user an user-friendly interface to interact with the *Two-level Transformation* Library. This is very useful in a lot of situations. In this paper I presented one example, migrating *XML* data to a SQL data base, but there are a lot of others examples.

Acknowledgments

Thanks to Flávio Ferreira and Hugo Pacheco for their help making *Two-level Transformation* working on GHC 6.6 and all the support

References

[BCPV06] Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, and Joost Visser. Coupled schema transformation and data conversion for xml and sql. Technical report, Departamento de Informática da Universidade do Minho, 2006.

[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.

[Gtk] Gtk2Hs. Gtk2hs is a gui for haskell based on gtk. http://haskell.org/gtk2hs.

[Hos] Kenneth Hoste. An introduction to gtk2hs, a haskell gui library.

[LFP06] Diogo Lapa, Flávio Ferreira, and Hugo Pacheco. Xml to sql conversion using type-safe two-level data transformation. Technical report, Departamento de Informática da Universidade do Minho, 2006.

A Text files

A.1 Example XML Schema

```
1 <?xml version="1.0"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3            targetNamespace="http://www.books.org"
4            xmlns="http://www.books.org"
5            elementFormDefault="qualified">
6
7   <xs:element name="Employees">
8     <xs:complexType>
9       <xs:choice minOccurs="1" maxOccurs="unbounded">
10        <xs:element name="Employee" type="EmployeeType"
    />
11        <xs:element name="Employee2" type="xs:string"/>
12      </xs:choice>
13    </xs:complexType>
14  </xs:element>
15  <xs:complexType name="EmployeeType">
16    <xs:sequence>
17      <xs:element name="FirstName" type="xs:string"
18                  minOccurs="1" maxOccurs="1"/>
19      <xs:element name="LastName" type="xs:string"
20                  minOccurs="1" maxOccurs="1"/>
21      <xs:element name="contact"type="xs:string"
22                  minOccurs="1" maxOccurs="unbounded"/>
23    </xs:sequence>
24    <xs:attribute name="ID" form="unqualified" type="xs:
    string"/>
25  </xs:complexType>
26 </xs:schema>
```

A.2 Example XML data instance

```
<?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>
    </Employee>
</Employees>
```

A.3 Example XML Schema parsed type

```
1 (Ann TypeAnn {fieldName = Just "Employees", key =
    Nothing, reference = []} (List (Either (Ann TypeAnn {
    fieldName = Just "Employee", key = Nothing, reference
     = []} (Prod (Prod (Ann TypeAnn {fieldName = Just "
    FirstName", key = Nothing, reference = []} String) (
    Prod (Ann TypeAnn {fieldName = Just "LastName", key =
     Nothing, reference = []} String) (List (Ann TypeAnn
    {fieldName = Just "contact", key = Nothing, reference
     = []} String)))) (Ann TypeAnn {fieldName = Just "
    att$ID", key = Nothing, reference = []} (Either
    String One)))) (Ann TypeAnn {fieldName = Just "
    Employee2", key = Nothing, reference = []} String))))
```

A.4 Example XML Schema parsed type after applying the rules

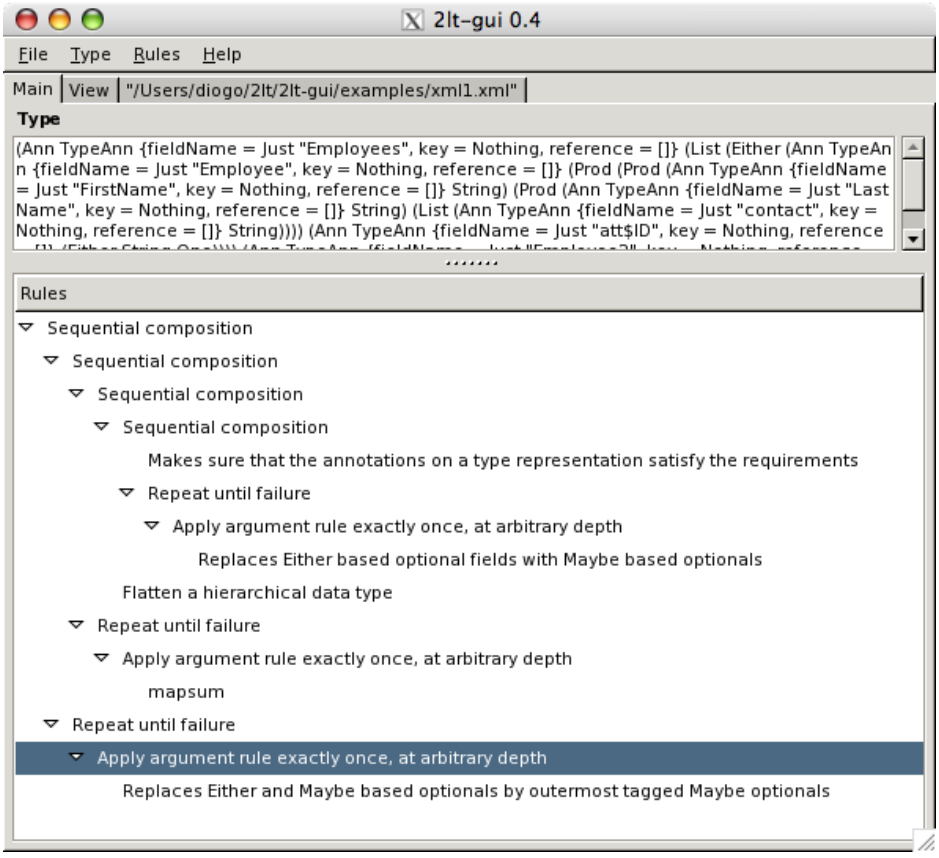
```
1 (Ann TypeAnn {fieldName = Nothing, key = Nothing,
    reference = []} (Prod (Ann TypeAnn {fieldName =
    Nothing, key = Nothing, reference = []} (Prod (Ann
    TypeAnn {fieldName = Just "Employees", key = Nothing,
     reference = []} (Map (Ann TypeAnn {fieldName = Just
    "index1", key = Just (1,[]), reference = []} Int) (
    Ann TypeAnn {fieldName = Nothing, key = Nothing,
    reference = []} (Prod (Ann TypeAnn {fieldName =
    Nothing, key = Nothing, reference = []} (Prod (Ann
    TypeAnn {fieldName = Just "Employee$FirstName", key =
     Nothing, reference = []} String) (Ann TypeAnn {
    fieldName = Just "Employee$LastName", key = Nothing,
    reference = []} String))) (Ann TypeAnn {fieldName =
    Just "Employee$att$ID", key = Nothing, reference =
    []} (Maybe String)))))) (Ann TypeAnn {fieldName =
    Just "Employee", key = Nothing, reference = []} (Map
    (Ann TypeAnn {fieldName = Nothing, key = Nothing,
    reference = []} (Prod (Ann TypeAnn {fieldName = Just
    "index1", key = Nothing, reference = [(Just "FK1"
    ,(1,[]))]} Int) (Ann TypeAnn {fieldName = Just "
    index2", key = Nothing, reference = []} Int))) (Ann
    TypeAnn {fieldName = Just "contact", key = Nothing,
    reference = []} String))))) (Ann TypeAnn {fieldName =
     Just "Employees1", key = Nothing, reference = []} (
    Map (Ann TypeAnn {fieldName = Just "index11", key =
    Nothing, reference = []} Int) (Ann TypeAnn {fieldName
     = Just "Employee2", key = Nothing, reference = []}
    String)))))
```

B Image files

B.1 Main tab

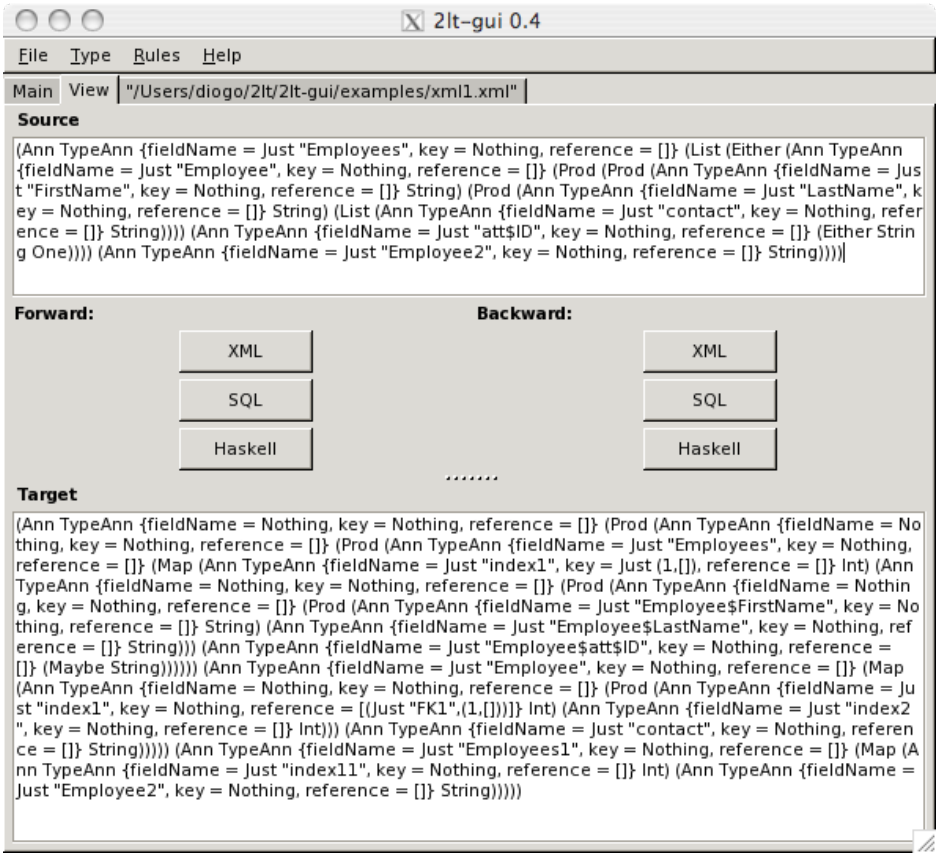
First tab showing the *TreeView* widget populated with rules and *TextView* showing an imported data type in the generic representation.

In this picture it's possible to see the hierarchical structure of the *TreeView* and the whole layout of this application.



B.2 View tab

Second tab showing the source and target types. In the middle we can see six buttons, three for forward transformations and the others three to backward transformations. Each one of this buttons opens a popup window requesting a file to parse the value according to the corresponding type: source type for forward transformation and target type for backward.



B.3 Result tab

Last tab showing the result: a couple of a type-level declaration and an instance of that type in value-level transformed according to the rules defined in the first tab.

After the user calculate this tab it's possible to export the data to three different types: *XML*, *SQL* and *Haskell*.