6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Refactoring via Database Representation*

Péter Diviánszky[a], Rozália Szabó-Nacsa[b], Zoltán Horváth[a]

[a]Department of Programming Languages and Compilers,
Eötvös Loránd University, Hungary
e-mail:divip@aszt.inf.elte.hu, hz@inf.elte.hu

[b]Department of Software Technology and Methodology,
Eötvös Loránd University, Hungary
e-mail: nacsa@inf.elte.hu

Abstract

We are going to develop such an interactive environment, where one can incrementally carry out programmer-guided meaning-preserving program transformations in functional languages. We discuss an alternative approach to the problems of extracting and storing the syntactic and also the static semantic information in order to be flexible enough to perform the desired transformations. In our approach the program to be redesigned is planed to be stored in a set of related abstract syntax- and semantic-ware tables (relational database).

During redesign process the programmer is faced with one of the selected "views" extracted from this database. Next to the traditional source code view programmer is served with different other views: view of "module hierarchy", view of "list of functions", process diagram, properties of functions etc. Functional language dialects (Clean, Haskell) can be seen as several views as well.

Different transformations can be carried out on different views, depending on which view is preferable for the programmer and/or which view is more suitable for the given transformation.

Categories and Subject Descriptors: D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments; D.3.2 [Programming Languages]: Language Classifications - *Applicative (functional) languages*;

Key Words and Phrases: Clean, Haskell, program transformation, refactoring, language-aware programming environments, semantic editors

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr. T037742 and by the Bolyai Research Scholarship.

1. Introduction

We are going to develop such an interactive environment, where one can incrementally carry out programmer-guided meaning-preserving program transformations in functional languages. We discuss an alternative approach to the problems of extracting and storing the syntactic and also the static semantic information in order to be flexible enough to perform the desired transformations. In our approach the program to be redesigned is planed to be stored in a set of related abstract syntax- and semantic-ware tables (relational database).

During redesign process the programmer is faced with one of the selected "views", extracted from this database. Essentially different views represent different abstraction levels. As far as the database stores all program related information, it allows easy access to accurate different type of semantic related information including static semantics, program properties, proof of properties, etc. Next to the traditional source code view programmer is served with different other views: view of "module hierarchy", view of "list of functions", process diagram, properties of functions etc. Functional language dialects (Clean, Haskell) can be seen as several views as well.

Different transformations can be carried out on different views, depending on which view is preferable for the programmer and/or which view is more suitable for the given transformation. One can perform the transformation highlighting some parts of the view and using the controls connected to the view. To ensure correct transformations, precondition of transformations will be checked with the help of the database tables. In some cases when the precondition does not hold, we will interactively offer compensation transformations to make it true letting the programmer to accept or refuse it. The modification is immediately recorded into the database, so the programmer can go on with the redesigned program.

In the future the function properties also can be stored and checked using the information stored in the database. It seems to be a natural tool for version management as well.

We believe that storing the program related syntactic and semantic information into a relational database results a simpler code in implementation of our transformation environment, and makes it easier to extend it's services.

2. Refactoring in Few Words

Refactoring is a programming technique for improving the design of a program without changing its behaviour. In other words, you clean up your code but do not change what it does. Refactoring may precede a program modification or extension, preparing the program for the modification, or may be used after finishing the work in order to bring the program into a nicer shape. The transformations of refactoring can be used for optimisation too. In these case the programmer writes a basic implementation ("specification") of the code, and improve its performance by refactoring steps.

Refactoring is carried out incrementally, in small steps, making small changes at once. After each step we must be sure, that the behavioural semantic of the code, at least from the "black box" point of view has not changed.

Refactoring is a particular sort of program transformation: renaming, unfolding, add/remove/split arguments, delete/add definition, etc. Consider a very simple refactoring: rename function. If you are going to make a "true refactoring", ie. behaviour-preserving transformations it is not enough to make a single name change, but you have to check if it can be accepted throughout all of your program, analysing bindings, calls, module and name space.

Refactoring is a well known technique within the object-oriented programming and software engineering communities, but it is not afloat within the functional programming one.

3. Database Representation of Programs

3.1. Overview

How can we store a program in a relational database? Our approach is the following: We translate the different program representations which can be found in an ordinary compiler into relations and store these relations in database tables.

One part of this representation is a straightforward translation of the syntax tree into relations. The identifiers are identified not by their name (a string) but by an id. For example, `a=b+c` can be translated as:

Id1 has name `a`
Id2 has name `b`
Id3 has name `c`
Id4 has name `+`

The root expression of function Id1 is expression Id5

Id5 is an application of function Id4

Id5 is an application whose 1st parameter is Id2
Id5 is an application whose 2nd parameter is Id3

Other database tables contains information about structure, types, properties and all type of information which may be useful during the transformation of the program.

The main advantages of this approach are:

Identification We can directly identify the fragments of the code like modules, functions, scopes, expressions and data structures by their ids. This is useful for interactive views. One solution for connecting the views and the database would be to store some positions related to the views in the database. Instead of this we will store the database ids in the views, so we can add a new view without changing the database.

Easy Access Given an id of an expression, we can easily reach not only its components but also its parent. (We can't do this in the syntax tree)

Arbitrary graph can be represented in a database — there is no difference between accessing a component from an expression and accessing the definition of a function from one of its application.

Clearer Representation The different aspects of the program — comments, names, syntactic polimorphism, structure, etc. — are separated clearly in different groups of tables, unlike in the syntax tree. Later on the program transformations can be classified by the property: which table groups will it modify.

Easier Compilation The executable can be generated easier from the database — the parsing phase and the connection between names and definitions vanish. The compiler also can use the transformations offered by our environment.

Communication between projects Concurrent refactoring of projects becomes possible because all of the projects will be in the same database.

A project will be not identified as set of modules but as a start function. In the other point of view, we will have only one big project and we have to maintain only this project. It will contain all of our old and new programs and they will always up to date.

Communication between programmers There is possibility for a central database.

Usage of Database Systems We should not bother with data storage problems; the database system which we use will solve these problems.

Flexible Treatment of Data With this mixed representation we will retrieve all type of information easily. On the other side the complexity of the system will grow. We would like solve this by a two level database, see section 4.

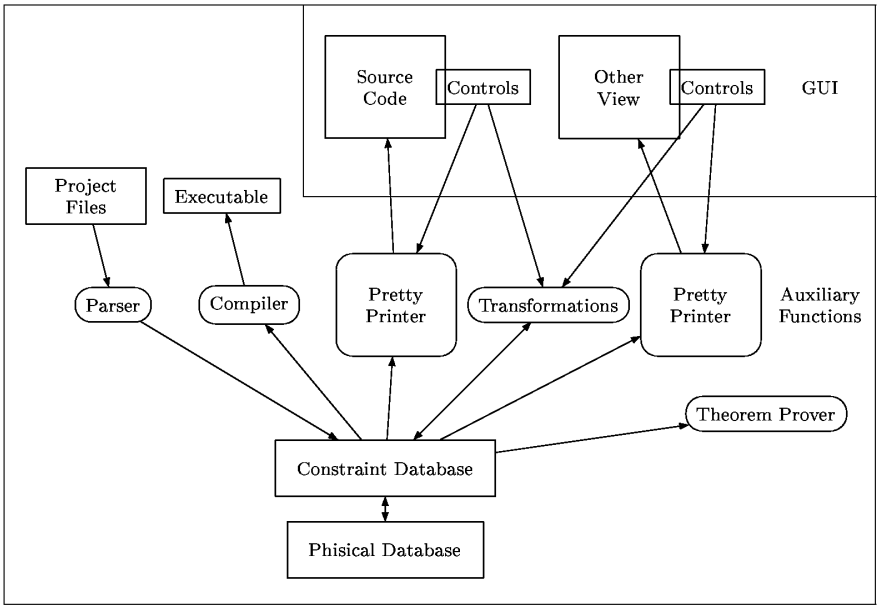
And the disadvantages are:

Indirect Access of data We can only reach the data through function calls so we can not benefit from the existence of patterns if we implement our functions in a functional language.

Execution Time Retrieving data from a database is slower than reaching data by memory pointers. However, searching in a tree data structure may be even slower.

4. System Architecture

Let us examine the system architecture from top down:



4.1. Graphical User Interface

We will offer several view of the database in the following way: For each view there is a pretty-printer program which produce an XML code from the high level database. The XML code will contain the description of the text/figure and it will contain some ids of the database for further identification.

For example in case of source-code view the XML code will contain the text, the font informations, the colouring, the indentation information and the ids of code fragments. A GUI program written in C++ with Qt will produce a scalable picture from the XML code. Those parts of the picture (the text) which has an id in the corresponding XML code may be selected by the user.

There will be several controls connected to these pictures. Several controls will change the picture but will not change the XML code (for example if the user changes the size of the window). Other controls will invoke the pretty printer with new parameters producing a new XML code and a new picture. Others again will invoke transformations with the id(s) of the selection. (Of course in this case the pretty printer also will be invoked.)

It seems to be natural to link the GUIs for the different views and the interface of code generator and the project adding tool in one GUI.

4.2. Auxiliary Functions

The auxiliary functions are the different pretty printers (one pretty printer per view), the transformations, the code generator and a parser by which we can add a Clean project to the database.

In the beginning the executable will be produced via the source code view by the Clean compiler. At the same time we would like to prepare the database for a future code generator.

4.3. Constraint Database

The auxiliary functions need different levels of information. If we just store all these information in database tables, the database would be very redundant, which is not good because it needs lots of memory and not maintainable.

For example, a row in one table would say that the expression of $id1$ is a function application of the function which name is f, while an other row in an other table would say that the same expression is a function application of the function of $id2$, and an other again that the function of $id2$ should have the name f. The last two assertions implies the first assertion, in other words we can give a constraint of the three tables which always holds.

We intend to fix the table names and types and the constraints. After it, we will write an interface for a physical database which fulfils the constraints and which is efficient enough. That would be our constraint database.

4.4. Low Level Database

We intend to use a Cleane interface for the MySQL implementation.

5. Related Ongoing Works

Beside refactoring, we would like to support two additional research projects concerning Clean.

At first we would like to integrate the functional theorem prover Sparkle [6] in our GUI. This could happen by storing properties of functions beside their types in the database and adding new views and controls for these properties and the `.sec` files of the Sparkle theorem prover.

The second related project is the introduction of abstract objects in Clean [7]. We can help to create the abstract objects in a suitable view and store its structure in the database. An other view would show these objects in a form (Core [6]) which can be processed by the Sparkle theorem prover.

References

[1] Li, H., Reinke, C., Thompson, S.: *Tool Support for Refactoring Functional Programs*, Haskell Workshop: Proceedings of the ACM SIGPLAN workshop on Haskell, Uppsala, Sweden, Pages: 27–38, 2003.

[2] Fóthi, Á., Horváth, Z., Nyéky-Gaizler, J.: *A Relational Model of Transformation in Programming*, Proceedings of the 3rd International Conference on Applied Informatics, Eger-Noszvaj, Hungary, Aug. 26-28, 1997. 335-349.

[3] Plasmeijer, R., Eekelen, M.: *Concurren Clean Language Report*, Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, 1998.

[4] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[5] *Martin Fowler's refactoring site*, www.refactoring.com

[6] de Mol, M., van Eekelen, m., Plasmeijer, R.: *SPARKLE: A Functional Theorem Prover*, International Workshop on the Implementation of Functional Languages, IFL 2001, Selected Papers, Springer-Verlag, LNCS 2312, pages 55-71.

[7] Horváth, Z., Kozsik, T., Tejfel, M.: *Verifying invariants of abstract functional objects — a case study* 6th International Conference on Applied Informatics, Eger, Hungary January 27-31, 2004.