Refactoring Erlang Programs

László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, Tamás Nagy
{lovei,hz,kto,viganiko,ntamas}@inf.elte.hu

Department of Programming Languages and Compilers
Eötvös Loránd University, Budapest

Introduction

Refactoring is a kind of source code transformation that preserves the meaning of the program. We use it to improve non-functional characteristics like readability or maintainability, or just because the next development step can be carried out more conveniently on some other form of the code.

Refactoring can be applied easily in projects built on the object-oriented paradigm, and software tools already exist to support safe refactoring. Functional languages, however, still lack proper tool support, only prototypes are available for pure functional languages, eg. Haskell.

Out goal is to create a refactoring toolset for the Erlang functional programming language. It should contain a framework that supports the condition checks necessary for refactoring and provides functionality for safe transformations, and can be integrated in a development environment easily. Here we present the basic approach to the implementation of this framework.

```
handle_call(add, {Pid, _T}, Store) ->
  {Id, NewStore} = add(Pid, Store),
  {reply, Id, NewStore}.

add(Pid, Store) ->
  Id = size(Store),
  {Id, insert(Id, Pid, Store)}.
```

Rename **Store** to **DB**

```
handle_call(add, {Pid, _T}, Store) ->
  {Id, NewStore} = add(Pid, Store),
  {reply, Id, NewStore}.

add(Pid, DB) ->
  Id = size(DB),
  {Id, insert(Id, Pid, DB)}.
```

Example: Rename a Variable

The most simple refactoring is renaming. To correctly rename something in a program, we need more than a simple find&replace: there are conditions to check to avoid name clash with existing identifiers, and semantic analysis is required to determine the exact scope of names.

The scoping rules of Erlang are simple, so the essence of variable renaming can be summarized as this:

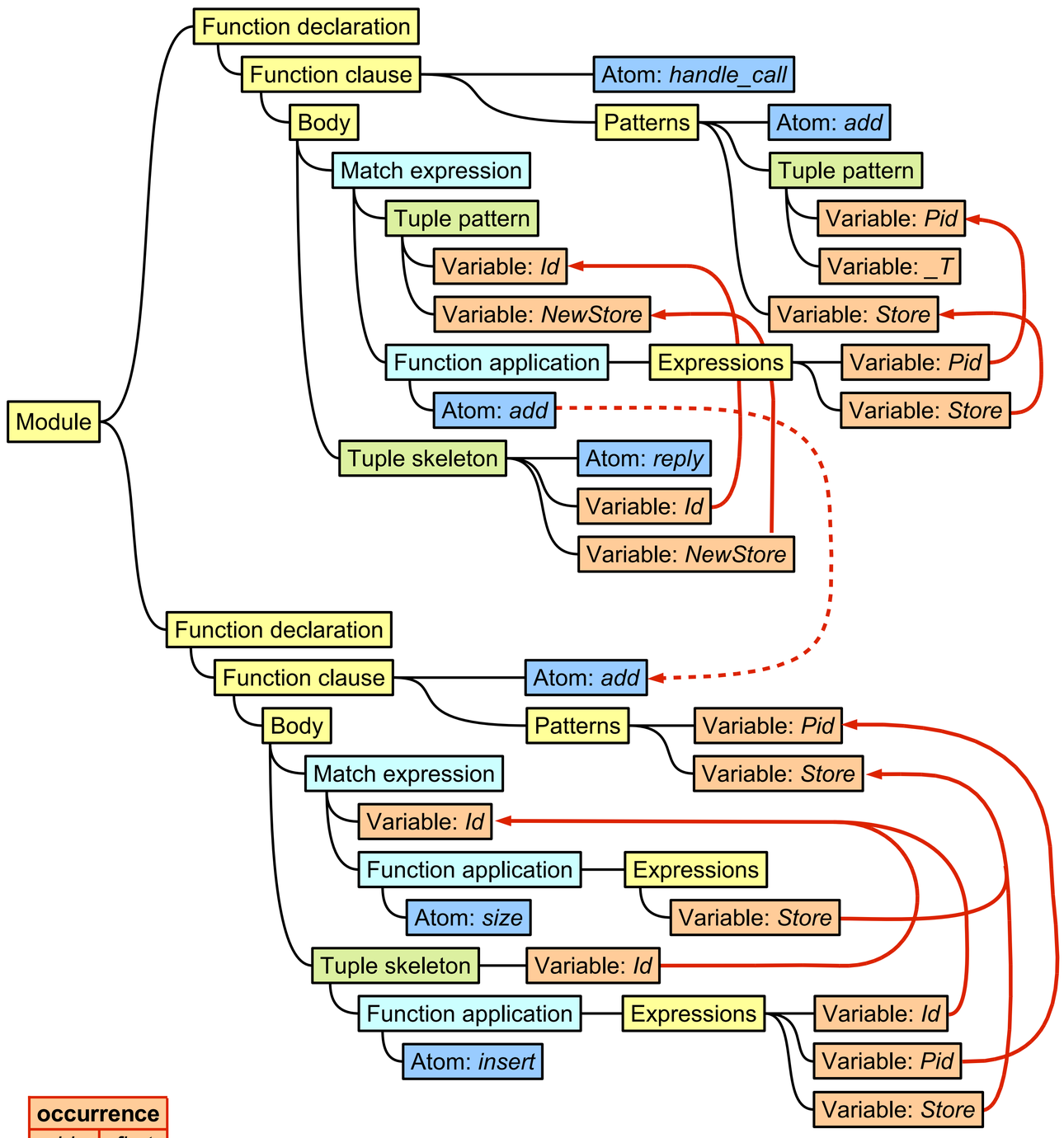
- the new variable name must nut occur in the body of the function that contains the variable, and
- the occurrences of the old name in the containing function body (and nowhere else) should be replaced with the new name.

In the example, we want to rename the **Store** parameter of the **add** function. The two rules mean that the new name cannot be **Pid** or **ID**, but can be **NewStore** or anything else (eg. **DB**), and the **Store** parameter of the function **handle_call** should not be modified.

Implementation of Refactoring

Every refactoring starts with the interpretation of the program text. Using standard techniques, this means that after parsing the source code, we get an abstract syntax tree (AST) as a result (it looks like the one on the right side, which corresponds to the example program text). The AST contains every information about the structure of the code, but many relations are not represented directly in it. For example, to rename a variable, we need to find every occurrence of it, and that requires the traversal of the tree, which is inefficient and complex to implement.

A helpful technique would be to store the needed links in the AST. To access every occurrence of a variable easily, we need to connect them. A possible way of this is to link them to a central point, eg. to the first occurrence in the AST. These “external” links are represented in the example by red arrows. If these are bidirectional links, obtaining every occurrence of a variable is trivial.



Relational Database Approach

Unfortunately, if we introduce external bidirectional links into a tree, it is not a tree anymore, and functional languages like Erlang are not efficient in solving generic graph related problems. Our special approach is to represent this graph as a set of relations in a relational database, and use SQL to manipulate it.

The tables below show the relevant part of the database that represents the example AST. Different types of nodes get their own tables, and every node has a globally unique identifier. Every link in the syntax tree is represented by a row in a table – this provides bidirectionality, because following a link (or a set of links) is done by selecting rows with a particular node ID.

External links are represented in the same way: rows in their own specific tables. By joining these tables, sophisticated subgraphs can be built easily, and selections with relatively simple conditions yield results that would need multiple tree traversals using other approaches. The variable renaming transformation is done by a single `update` statement, and with the help of a table that contains variable visibility information for functions, the condition check has the same complexity.

application

id	*pos*	*arg*
21	0	13
21	1	110
21	2	111
22	0	16
22	1	113
23	0	17
23	1	115
23	2	116
23	3	117

match

id	*pos*	*arg*
41	1	32
41	2	21
42	1	112
42	2	22

variable

id	*name*
101	Pid
102	_T
103	Store
104	Id
105	NewStore
106	Pid
107	Store
108	Id
109	NewStore
110	Pid
111	Store
112	Id
113	Store
114	Id
115	Id
116	Pid
117	Store

occurrence

id	*first*
101	101
102	102
103	103
104	104
105	105
106	101
107	103
108	104
109	105
110	110
111	111
112	112
113	111
114	112
115	112
116	110
117	111

atom

id	*name*
11	handle_call
12	add
13	add
14	reply
15	add
16	size
17	insert

tuple

id	*pos*	*elem*
31	1	101
31	2	102
32	1	104
32	2	105
33	1	14
33	2	108
33	3	109
34	1	114
34	2	23

Prototype tool

A working prototype built on these principles gives us some basis to evaluate the presented approach. The tool is implemented in Erlang using ODBC to communicate with a MySQL database, and it is integrated into Emacs to provide a standard interface.

The advantages of the approach are mostly implementation-related. As sets of links in the graph can be manipulated easily an efficiently by relatively simple SQL statements, the implementation of a particular refactoring is easy, and consecutive steps are executed quickly.

On the other hand, the complexity is shifted to the construction of the database, which means slower and more complex program analysis. However, in case of large systems with lots of modules, and using carefully selected guided editor capabilities that manipulate the database directly, this drawback can be minimalised. An other disadvantage is the dependency on a database management system, but this is not a big problem because free database systems are available.

Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary and ELTE CNL