Refactoring Erlang Programs *

Huiqing Li, Simon Thompson
University of Kent, UK

László Lövei, Zoltán Horváth, Tamás Kozsik, Anikó Víg, Tamás Nagy
Eötvös Loránd University, Hungary

Abstract

We describe refactoring for Erlang programs, and work in progress to build two tools to give machine support for refactoring systems written in Erlang. We comment on some of the peculiarities of refactoring Erlang programs, and describe in some detail a number of refactorings characteristic of Erlang.

1. Introduction

Refactoring [6] is the process of improving the design of a program without changing its external behaviour. Behaviour preservation guarantees that refactoring does not introduce (or remove) any bugs. Separating general software updates into functionality changes and refactorings has well-known benefits. While it is possible to refactor a program by hand, tool support is invaluable as it is more reliable and allows refactorings to be done (and undone) easily. Refactoring tools can ensure the validity of refactoring steps by automating both the checking of the conditions for the refactoring and the refactoring transformation itself, making the process less painful and error-prone.

Refactoring has been applied to a number of languages and paradigms, but most of the work in building tools has concentrated on object-oriented programming. In this paper we report on work in progress at our universities to build tools to support the refactoring of Erlang programs.

The paper begins with a brief introduction to refactoring, which is followed by a discussion of the particular question of refactoring Erlang systems. We then describe the approaches taken by our two teams: in a nutshell, the Kent team work over a enriched abstract syntax tree (AST), whereas the research at Eötvös Loránd University builds the representation in a relational database.

After describing the systems we speculate on what refactorings are the most appropriate to Erlang and are most useful to the working Erlang programmer, before concluding and surveying future work for both teams.

2. Refactoring

Refactorings transform the structure of a program with out changing its functionality. They are characterised by being *diffuse* and *bureaucratic*. They ate diffuse in the sense that a typical refactoring will affect the whole of a module or set of modules, rather than a single definition in a program, which is often the case for a program optimising transformation. They are bureaucratic in that they require attention to detail; for instance, taking into account the binding structure of a program.

Refactorings are not simply syntactic. In order to preserve the functionality of a program, refactorings require awareness of various aspects of the semantics of the program including types and module structure and most importantly the *static semantics* of the program: that is the scope of definitions, the binding structure of the program (the association between the use of an identifier and its definition), the uniqueness of definitions and so forth.

Each refactoring comes with a set of side conditions, which embody when a refactoring can be applied to a program without changing its meaning. Our experience of building refactoring tools so far shows that for most refactorings, the side-condition analysis is more complex than the program transformation part. Taking

* Supported by EPSRC in the UK, GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK, Ericsson Hungary, Bolyai Research Fellowship and ELTE CNL in Hungary

a concrete example, the general side conditions for renaming an identifier could be as follows.

The existing binding structure should not be affected. No binding for the new name may intervene between the binding of the old name and any of its uses, since the renamed identifier would be captured by the renaming. Conversely, the binding to be renamed must not intervene between bindings and uses of the new name.

These side-conditions apply to most programming languages. However, each programming language may also impose its own particular constraints on this refactoring. For example, in an Erlang program using the OTP library, a user should not rename certain functions exported by a call-back module. For some languages, refactoring conditions can be checked at compile time; the more dynamic nature of Erlang means that some necessary conditions can only decided at run-time; we return to this point below.

2.1 Tool Support for Refactorings

Although it is possible to refactor a program manually, it would be both tedious and error-prone to refactor large programs this way. Interactive tool support for refactoring is therefore necessary, as it allows refactorings to be performed easily and reliably and to be undone equally easily.

A refactoring tool needs to get access to both the syntactic and static semantic information of the program under refactoring. While detailed implementation techniques might be different, most refactoring tools go through the following process: first transform the program source to some internal representation, such as an abstract syntax tree (AST) or database table; then analyse the program to extract the necessary static semantic information, such as the binding structure of the program, type information and so forth.

After that, program analysis is carried out based on the internal representation of the program and the static semantics information to validate the side-conditions of the refactoring. If the side-conditions are not satisfied, the refactoring process stops and the original program is unchanged, otherwise the internal representation of the program is transformed according to the refactoring. Finally, the transformed representation of the program need to be presented to the programmer in program source form, with comments and the original program appearance preserved as much as possible.

```
-module (sample).
-export([printList/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.
```

Figure 1. The initial program

```
-module (sample).
-export([printList/1, broadcast/1]).

printList([H|T]) ->
    io:format("~p\n", [H]),
    printList(T);
printList([]) -> true.

broadcast([H|T]) ->
    H ! "The message",
    broadcast(T);
broadcast([]) -> true.
```

Figure 2. Adding a new function naïvely

The Kent group are responsible for the project 'Refactoring Functional Programs' [7], which has developed the Haskell Refactorer, HaRe [9], providing support for refactoring Haskell programs. HaRe is a mature tool covering the full Haskell 98 standard, including "notoriously nasty" features such as monads, and is integrated with the two most popular development environments for Haskell programs: Vim and (X)Emacs. HaRe refactorings apply equally well to single- and multiple-module projects. HaRe is itself implemented in Haskell.

Haskell layout style tends to be idiomatic and personal, especially when a standard layout is not enforced by the program editor, and so needs to be preserved as much as possible by refactorings. HaRe does this, and also retains comments, so that users can recognise their source code after a refactoring. The current release of HaRe supports 24 refactorings, and also exposes an API [10] for defining Haskell refactorings and program transformations.

3. Refactoring Erlang Programs

Figures 1 - 5 illustrate how refactoring techniques can be used in the Erlang program development process.

```
-module (sample).
-export([printList/1]).

printList(L) ->
    printList(fun(H) ->
        io:format("~p\n", [H]) end, L).

printList(F,[H|T]) ->
    F(H),
    printList(F, T);
printList(F,[]) -> true.
```

Figure 3. The program after generalisation

```
-module (sample).
-export([printList/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 4. The program after renaming

The example presented here is small-scale, but it is chosen to illustrate aspects of refactoring which can scale to larger programs and multi-module systems.

In Figure 1, the function `printList/1` has been defined to print all elements of a list to the standard output. Next, suppose the user would like to define another function, `broadcast/1`, which broadcasts a message to a list of processes. `broadcast/1` has a very similar structure to `printList/1`, as they both iterate over a list doing something to each element in the list. Naïvely, the new function could be added by copy, paste, and modification as shown in Figure 2. However, a *refactor then modify* strategy, as shown in Figures 3 - 5, would make the resulting code easier to maintain and reuse.

Figure 3 shows the result of generalising the function `printList` on the sub-expression

```
io:format("~p/n",~[H])
```

The expression contains the variable `H`, which is only in scope within the body of `printList`. Instead of generalising over the expression itself, the transformation is achieved by first abstracting over the free variable `H`, and by making the generalised parameter a *function* `F`. In the body of `printList` the expression `io:format("~p/n", H])` has been replaced with `F` applied to the local variable `H`.

```
-module (sample).
-export([printList/1, broadcast/1]).

printList(L) ->
    forEach(fun(H) ->
        io:format("~p\n", [H]) end, L).

broadcast(Pids)->
    forEach(fun(H) ->
        H ! "The message" end, Pids).

forEach(F,[H|T]) ->
    F(H),
    forEach(F, T);
forEach(F,[]) -> true.
```

Figure 5. The program after adding a function

The arity of the `printList` has thus changed; in order to preserve the interface of the module, we create a new function, `printList/1`, as an application instance of `printList/2` with the first parameter supplied with the function expression:
`fun(H) -> io:format("~p/n", [H]) end.`
Note that this transformation gives `printList` a functional argument, thus making it a characteristically 'functional' refactoring.

Figure 4 shows the result of renaming `printList/2` to `forEach/2`. The new function name reflects the functionality of the function more precisely. In Figure 5, function `braodcast/1` is added as another application instance of `forEach/2`.

Refactorings to generalise a function definition and to rename an identifier are typical structural refactorings, implemented in our work on both Haskell and Erlang.

3.1 Language Issues

In working with Erlang we have been able to compare our experience with what we have done in writing refactorings for Haskell. Erlang is a smaller language than Haskell, and in its pure functional part, very straightforward to use. It does however have a number

of irregularities in its static semantics, such as the fact that it is possible

- to have multiple defining occurrences of identifiers, and
- to nest scopes, despite the perception that there is no shadowing of identifiers in Erlang.

Erlang is also substantially complicated by its possibilities of reflection: function names, which are atoms, can be computed dynamically, and then called using the `apply` operator; similar remarks apply to modules. Thus, in principle it is impossible to give a complete analysis of the call structure of an Erlang system statically, and so the framing of side-conditions on refactorings which are both necessary and sufficient is impossible.

Two solutions to this present themselves. It is possible to frame *sufficient* conditions which prevent dynamic function invocation, hot code swap and so forth. Whilst these conditions can guarantee that behaviour is preserved, they will in practice be too stringent for the practical programmer. The other option is to *articulate* the conditions to the programmer, and to pass the responsibility of complying with them to him or her. This has the advantage of making explicit the conditions without over restricting the programmer through statically-checked conditions. It is, of course, possible to insert assertions into the transformed code to signal condition transgressions.

Compared to Haskell users, Erlang users are more willing to stick to the standard layout, on which the Erlang Emacs mode is based. Therefore a pretty-printer which produces code according to the standard layout is more acceptable to Erlang users.

3.2 Infrastructure Issues

A number of tools support our work with Erlang. Notable among these is the `syntax-tools` package which provides a representation of the Erlang AST within Erlang. The extensible nature of the package allows syntax trees to be equipped with additional information as necessary. For example, Erlang Syntax Tools provides functionalities for reading comment lines from Erlang source code, and for inserting comments as attachments on the AST at the correct places; and also the functionality for pretty printing of abstract Erlang syntax trees decorated with comments.

The *Distel* infrastructure helps us to integrate refactorings with Emacs, and thus make them available within the most popular Erlang IDE.

4. Our Approaches

Both University of Kent and Eötvös Loránd University are now in the process of building a refactoring tool for Erlang programs, however different techniques have been used to represent and manipulate the program under refactoring. The Kent approach uses the *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs, and program analysis and transformation manipulate the AASTs directly; whereas the Eötvös Loránd approach uses relational database, MySQL, to store both syntactic and semantic information of the Erlang program under refactoring, therefore program analysis and transformation are carried out by manipulating the information stored in the database.

One thing that is common between the two refactoring tools is the interface. Both refactoring tools are embedded in the Emacs editing environment, and both make use of the functionalities provided by Distel [8], an Emacs-based user interface toolkit for Erlang, to manage the communication between the refactoring tool and Emacs.

In this section, we first illustrate the interface of the refactoring tools, explain how a refactoring can be invoked, then give an overview of the two implementation approaches. A preliminary comparison of the two approaches follows.

4.1 The Interface

While the catalogue of supported refactorings is slightly different at this stage, the interfaces of the two refactoring tools share the same look and feel. In this paper, we take the refactoring tool from University of Kent as an example to illustrate how the tool can be used.

A snapshot of the Erlang refactorer, which is called *Wrangler*, is shown in Figure 6. To perform a refactoring, the source of interest has to be selected in the editor first. For instance, an identifier is selected by placing the cursor at *any* of its occurrences; an expression is selected by highlighting it with the cursor. Next, the user chooses the refactoring command from the *Refactor* menu, which is a submenu of the *Erlang* menu, and input the parameter(s) in the mini-buffer if prompted.

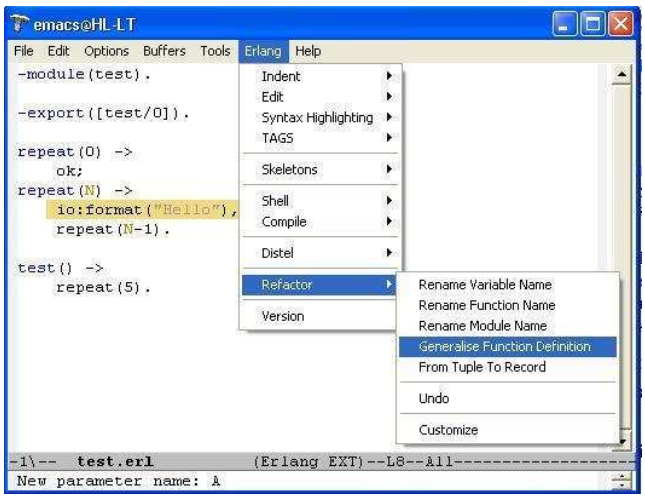


Figure 6. A snapshot of Wrangler

After that, the refactorer will check the selected source is suitable for the refactoring, and the parameters are valid, and the refactoring's side-conditions are satisfied. If all checks are successful, the refactoring will perform the refactoring and update the program with the new result, otherwise it will give an error message and abort the refactoring with the program unchanged.

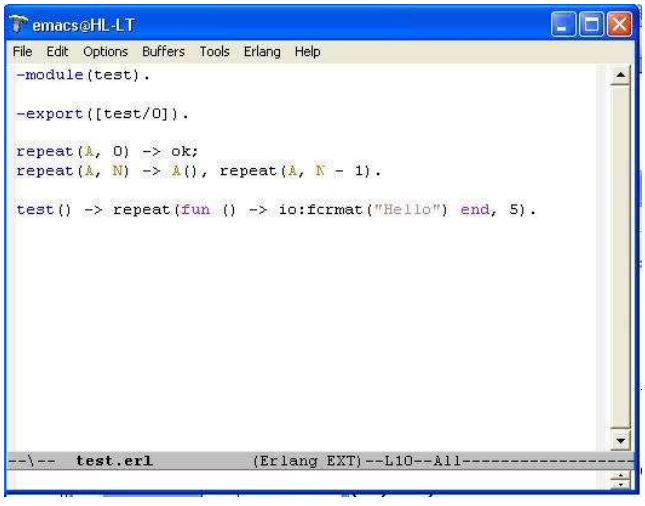


Figure 7. A snapshot of Wrangler showing the result of generalising a definition

Figure 6 shows a particular refactoring scenario. The user has selected the expression `io:format("Hello")` in the definition of `repeat/1`, has chosen the *Generalise Function Definition* command from the *Refactor* menu, and is just entering a new parameter name `A` in the mini-buffer. After this, the user would press the *Enter* key to perform the refactoring. The result of this refactoring is shown in Figure 7: the new parameter `A` has been added to the definition of `repeat/1`, which now becomes `repeat/2`, and the selected expression, wrapped in a fun-expression because of the side-effect problem, is now supplied to the call-site of the generalised function as an actual parameter.

All the implemented refactorings are module-aware. In the case that a refactoring affects more than one module in the program, a message telling which unopened files, if there is any, have been modified by the refactorer will be given after the refactoring has been successfully done. The *customize* command from the *Refactor* menu allows the user to specify the boundary of the program, i.e. the directories that will be searched and analysed by the refactorer.

Undo is supported by the refactorer. Applying *undo* once will revert the program back to the status right before the last refactoring performed.

4.2 The Kent Approach

In this approach, the refactoring engine is built on top of the infrastructure provided by SyntaxTools [1]. It uses the *annotated abstract syntax tree* (AAST) as the internal representation of Erlang programs; both program analysis and transformation manipulate the AASTs directly.

4.2.1 The SyntaxTools Package

After having investigated the few available Erlang frontends, we decided to build our refactoring tool on top of the infrastructure provided by SyntaxTools. SyntaxTools is a library from the Erlang/OTP release. This library contains modules for handling Erlang abstract syntax trees (ASTs), in a way that is compatible with the "parse trees" of the standard library module `erl_parse`, together with utilities for reading source files in unusual ways, e.g. bypassing the Erlang preprocessor, and pretty-printing syntax trees. The data types of the abstract syntax is nicely defined so that the nodes in an AST have an uniform structure, and their types are context-independent. We chose to build our refactoring tool on top of SyntaxTools for the following reasons:

- The uniform representation of AST nodes and the type information (more precisely, the syntax category of the syntax phrase represented by the node) stored in each AST node allow us to write generic functions that traverse into subtrees of an AST while

treating most nodes in an uniform way, but those nodes with a specific type in a specific way. This is a great help as both program analysis and transformation involve frequent AST traversals.

- The representation of AST nodes allows users to add their own annotations associated with each AST node. The annotation may be any terms. This facility can be used by the refactorer to attach static semantic information, or any necessary information, to the AST nodes.
- SyntaxTools also contains functionalities for attaching comments to the ASTs representing a program, and a pretty-printer for printing Erlang ASTs attached with comments. This liberates us from the comment-preservation problem, which is also critical for a refactoring tool to be usable in practice. While a pretty-printer could produce a program that has a slightly different appearance with the original one, this does not seem to be a big problem in a community where people tend to accept and use the standard program layout rules.

4.2.2 Adding Static Semantics, Locations, and Type Information

SyntaxTools provides the basic infrastructure for source-to-source Erlang program transformation, even some utility functions for free/bound variable analysis, AST traversals, etc, to ease the analysis of source code structure. However, in order to make program analysis, transformation, as well as the mapping from textual presentation of a syntax phrase to its AST presentation easier, we have annotated the ASTs produced by SyntaxTools with even more information, therefore comes the *Annotated Abstract Syntax Tree* (AAST). What follows summaries the main information we have added, or are trying to add, to the Erlang AST.

- Binding information. The binding information of variables and function names is annotated in the AST in terms of defining and use locations. For example, each occurrence of a variable node in the AST is annotated with its occurrence location in the source, as well as the location where it is defined. Locations are presented by the combination of filename, line number, and column number (the standard Erlang lexer and parser had to be modified in order to get the column number). With the binding information, we can easily check whether two variable/function names refer to the same thing just by looking at their defining locations.
- Range information. Each AST node is annotated with its start and end location in the source code. This makes it easier to map a syntax phrase selected from the textual representation in the editor to its AST representation.
- Category information. The original abstract Erlang syntax does distinguish different kinds of syntax categories, such as functions, attributes, if-expressions, etc. The category information introduced here is mainly to distinguish expressions from patterns.
- Type information. Type information is necessary for some refactorings, and some refactorings require even more refined information than the basic data types defined in Erlang. For example, suppose the same atom, `foo` say, is used as both a module name and a function name in the program. In this case, renaming the module name `foo` may need to know whether an occurrence of the atom `foo` refers the module name being renamed or the function name; therefore, simply knowing the type of `foo` is *atom* is not enough. Adding type information to the AST is currently work-in-progress; we are investigating whether the functionalities provided by TypEr [11], a type annotator for Erlang code, can be used to retrieve the necessary type information.

4.2.3 The Implementation Architecture

Figure 8 summaries the implementation architecture of Wrangler.

Figure 8. The Implementation Architecture

4.3 The Eötvös Loránd Approach

Instead of annotating the ASTs with information that is necessary for program analysis and transformation, in this approach, we use a relational database, MySQL, to store both abstract Erlang syntax trees and the associated static semantic information, and use SQL to manipulate the stored information. This approach is mainly influenced by the experience from refactoring Clean programs [17, 4].

In the relational database representation, there are two kinds of tables: tables that store the AST, and tables

```
gcd30(N15, M16) when N17 >=18 M19 →
    gcd23( N24 −15 M26, M28);
```

Figure 9. Source code of the example function clause.

information in the AST	database equivalent	
	table name	record in that table
1st parameter of clause 30 is node 15	clause	30, 0, 1, 15
the name of variable 15 is N	name	15, "N"
2nd parameter of clause 30 is node 16	clause	30, 0, 2, 16
clause 30 has a guard, node 22	clause	30, 1, 1, 22
the left and right operands and the operator of the infix expression 20 are nodes 17, 19 and 18, respectively	infix_expr	20, 17, 18, 19
the body of clause 30 is node 29	clause	30, 2, 1, 29
application 29 applies node 23	application	29, 0, 23
the content of atom 23 is gcd	name	23, "gcd"
1st param. of application 29 is node 27	application	29, 1, 27

Table 1. The representation of the code in Figure 9 in the database.

that store semantic information. The syntax-related tables correspond to the "node types" of the abstract syntax of Erlang as introduced in the Erlang parser. Semantic information, such as scope and visibility of functions and variables, is stored separately in an extensible group of tables. Adding a new feature to the refactoring tool requires the implementation of an additional semantic analysis and the construction of some tables storing the collected semantic information. It is possible to store semantic information of different levels of abstraction in the same database and to support both low-level and high-level transformations.

As an example, consider the code in Figure 9. This is one of the clauses of a function that computes the greatest common divisor of two numbers. Each node of the AST is given a unique id. Every module also has its own id. These ids are written as subscripts in the code.

The database representation of the AST is illustrated in Table 1. The table names *clause*, *name*, *infix_expr* and *application* refer to the corresponding syntactic categories. Without addressing any further technical details, one can observe that each table relates parent nodes of the corresponding type with their child nodes.[1]

In order to make information retrieval faster, an auxiliary table, *node_type* was introduced. This table binds the id of each parent node to the table corresponding to its type. Semantic information about Erlang programs are stored in tables such as *var_visib*,*fun_visib* and *fun_def*. The table *var_visib* stores visibility information on variables, namely which occurrences of a variable name identify the same variable. This table has two columns: *occurrence* and *first_occurrence*. The former is the identifier of a variable occurrence, and the latter is the identifier of the first occurrence of the same variable. The table *fun_visib* stores similar information for function calls, and *fun_def* maintains the arity and the defining clauses of functions.

The *rename variable* and *rename function* transformation is supported with three further table, *forbidden_names*, *scope* and *scope_visib*. The first describes names that are not allowed to use for variables (and for functions). This table contains the reserved words in Erlang, names of the built-in functions, and also user-specified forbidden names. The *scope* table contains the scope of the nodes, what is the most inner scope they are in. The *scope_visib* table stores the hierarchy of the scopes.

As you can observe the resulting data structure is not a tree, but rather a graph, which represents more general connections. We hope it makes easier to implement the refactor steps.

4.3.1 The Implementation Architecture

Figure 10 summaries the implementation architecture of this approach. The refactor step updates the

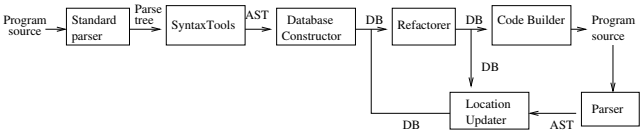


Figure 10. The Implementation Architecture

database (which represents the AST and the semantic information), but the position information might

[1] The price for the separation of tables containing syntactic information from tables containing semantic information is an increased redundancy in the database. For example, the "names" table stores the variable name for each occurrence of the same variable.

no longer reflect the actual positions in the program source. In order to keep the position information up-to-date, we build up the updated syntax tree from the database and use the pretty-printer to refresh the code, then the position information is updated by a simultaneous traversal of the syntax tree represented in the database, and the AST generated by parsing the refreshed code.

4.4 Comparison

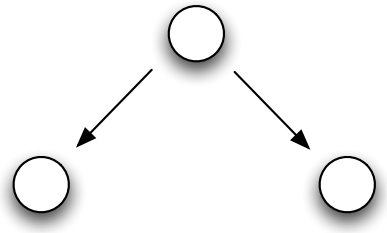
The major difference between the two approaches lies in how the syntactic and semantic information is stored and manipulated. Our first impression is that the second approach needs more time and effort on database designing and the migration of information from abstract Erlang syntax trees to the database; whereas the first approach is relatively light-weight. However, as the second approach tries to avoid reconstruction of the database between two consecutive refactorings by incrementally updating the database so as to keep the stored syntactic and semantic information up-to-date, it may worth the effort. At this stage, it is hard to say which approach is better.

Once both of the two refactoring tools have had support for a number of representative, module-aware refactorings, we would like to test and compare them on some large-scale Erlang programs, and find out the pros and cons of each approach.

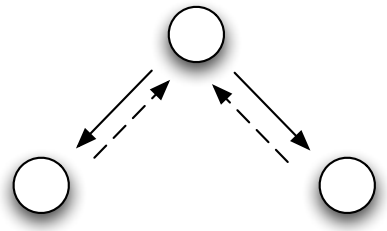
5. Refactorings: The Next Step

The refactorings implemented by both teams thus far are *structural* by nature; we plan also to implement module and data refactorings in line with our work in HaRe. We are also investigating transformations of features characteristic to Erlang, In this section we look at one example, which changes the pattern of communication within a system. We first present a scenario.

A system is constructed in which communication between processes is asynchronous; that is, messages are sent and receipts are not required. It becomes possible to optimise processing within the network by chopping out whole sections; this, however, requires sending a reply back to the sender. As is the case in many software developments, a refactoring can be the first step in modifying the system; in this case, the first step is to make the communication *synchronous*. In pictures, one way communication



is replaced by a two-way, synchronous pattern:



Such a transformation requires a message send to be followed by a receipt, transforming

```
pid!{self(),msg}
```

to

```
pid!{self(),msg},
receive
    {pid, ok}-> ok
```

and in the recipient the code

```
receive {Parent,msg} -> body
```

is replaced by

```
receive {Parent,msg} ->
    Parent!{self(),ok},
    body
```

We envisage implementing other concurrency-related refactorings, and in particular we expect to support transformations of concurrent systems written within the OTP framework; we discuss some other Erlang-specific refactorings now.

Built-in support for concurrency is one of the main features of Erlang. In a well-designed Erlang program, there should be a one-to-one mapping between the number of parallel processes and the number of truly parallel activities in the real world. The following refactoring allows to adjust the process structure in a program.

- *Introduce/remove concurrency* by introducing or removing concurrent processes so as to achieve a better mapping between the parallel processes and the truly parallel activities of the problem being solved. For example, using processes and message passing when a function call can be used instead is a bad programming practice, and this refactoring should help to eliminate the un-desired process and message passing with a function call.

While defensive-style programming is a good programming practice when a sequential programming language is used, non-defensive style programming is the right thing to do when programming with Erlang. Erlang's *worker/supervisor* error handling mechanism allows a clear separation of error recovery code and normal case code. In this mechanism, both *workers* and *supervisors* are processes, where *workers* do the job, and *supervisors* observe the *workers*. If a *worker* crashes, it sends an error signal to its *supervisor*.

- *From defensive-style programming to non-defensive style*. This refactoring helps to transform defensive-style sequential error-handling code written in Erlang into concurrent error handling, typically using supervisor trees.

Erlang programming idioms also expose various refactoring opportunities. Some examples are:

- *Transform a non-tail-recursive function to a tail-recursive function*. In Erlang, all servers must be tail-recursive, otherwise the server will consume memory until the system runs of it.
- *Remove import attributes*. Using import attributes makes it harder to directly see in what module a function is defined. Import attributes can be removed by using remote function call when a call of function defined in another module is needed.
- *From meta to normal function application* by replacing `apply(Module, Fun, Args)` with

  ```
  Module:Fun(Arg1, Arg2,..., ArgN)
  ```

  when the number of elements in the arguments, `Args`, is known at compile-time.
- *Refactoring non-OTP code towards an OTP pattern*. Doing this from pure Erlang code is going to be very challenging, but the whole transformation can be decomposed into a number of elementary refactorings, and each elementary refactoring brings the code a bit closer to the desired OTP pattern.

6. Conclusion

We conclude by surveying related work, and by looking at what we plan to do next.

6.1 Related Work

Programmers used refactoring to make their code more readable, better structured or more apt for further extensions long before the first papers appeared on the topic (e.g. [13]). The field was given much greater prominence with the publication by Fowler's [6], which particularly addressed a wide range of 'manual' refactorings for Java.

Tool support for refactoring is available mostly to object-oriented languages. The first tool was the refactoring browser for Smalltalk [16]. Most tools target Java (IntelliJ Idea, Eclipse, JFactor, Together-J etc.), but there are some for .NET (ReSharper, C# Refactory, Refactor! Pro and JustCode!), C++ (SlickEdit, Ref++ and Xrefactory) and other languages as well. Common refactorings offered by the tools include those that rename program entities (variables, subprograms, modules), those that extract or inline program units, or those that change the static model of classes. A good summary of tools and refactorings can be found at [5], and [12] is an exhaustive survey of the field of software refactoring.

Marcio Lopes Cornèlio formalizes refactorings in an object-oriented language [2]. Some preconditions of refactorings are not simple to compute from the static program text in case of dynamic languages like Smalltalk and Python [16, 15]. The Smalltalk refactoring browser applies dynamic analysis to resolve this problem.

To improve the quality of a code according to a redesign proposal or enforce coding conventions needs support for complex refactoring operations. Planning a sequence of refactoring steps needs refactoring analysis and plan to achieve desirable system structure [14]. Frameworks and libraries change their APIs from time to time. Migrating an application to a new API is tedious work, but typically some eighty percent of the changes will be refactoring steps. Automated detection, record and replay of refactoring steps may support upgrading of components according using the new API [3].

6.2 Future Work

It is a short-term goal for the teams to contrast their approaches on example code bases, to compare the utility of the two approaches. For instance, the ADT approach has the advantage of being more lightweight, but the database representation can offer versioning of code and the concurrent handling of refactoring steps in some cases.

In the medium term, each team will build support for further refactorings, particularly those supporting practising Erlang programmers. In particular we will build refactorings to support the transformation of data representations, changes to patterns of concurrent communication and integration with the OTP framework.

In the longer term we look forward to machine-supported refactoring becoming a valuable part of the Erlang programmers' toolkit.

References

[1] Carlsson, R. . Erlang Syntax Tools. `http://www.erlang.org/doc/doc-5.4.12/lib/syntax_tools-1.4.3/doc/html/`.

[2] Cornèlio, M.L.: Refactorings as Formal Refinements, PhD thesis, Universidade Federal de Pernambuco, 2004.

[3] Dig, D.: Toward Automatic Upgrading of Component-Based Applications, ECOOP 2006 Doctoral Symposium and PhD Students Workshop, Nantes, Fance, 2006. `http://www.ecoop.org/phdoos/ecoop2006ds/`.

[4] Diviánszky, P. and Szabó-Nacsa, R. and Horváth, Z. Refactoring via Database Representation. In L. Csőke, P. Olajos, P. Szigetváry, and T. Tómács, editors, *The Sixth International Conference on Applied Informatics (ICAI 2004), Eger, Hungary*, volume 1, pages 129–135, 2004.

[5] Fowler, M.: Refactoring Home Page, `http://www.refactoring.com/`.

[6] Fowler, M. *et. al.*, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] Refactoring Functional Programs, `http://www.cs.kent.ac.uk/projects/refactor-fp/`.

[8] Gorrie, L.. Distel: Distributed Emacs Lisp (for Erlang).

[9] Li, H. and Reinke, C. and Thompson, S., Tool Support for Refactoring Functional Programs in, ACM SIGPLAN Haskell Workshop 2003, Uppsala, Sweden, Johan Jeuring (ed.), 2003.

[10] Li, H. and Reinke, C. and Thompson, S., The Haskell Refactorer, HaRe, and its API., Electr. Notes Theor. Comput. Sci., 141 (4), 2005.

[11] Lindahl, T. and Sagonas, K. F.. TypEr: a Type Annotator of Erlang Code. In *ACM SIGPLAN Erlang Workshop 2005*, 2005.

[12] Mens, T. and Tourwé, T., A Survey of Software Refactoring, IEEE Trans. Software Eng., 30 (2), 2004.

[13] Opdyke, W.: Refactoring Object-Oriented Frameworks, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[14] Perez, J. Overview of the Refactoring Discovering Problem, ECOOP 2006 Doctoral Symposium and PhD Students Workshop, Nantes, Fance, 2006. `http://www.ecoop.org/phdoos/ecoop2006ds/`.

[15] Adventures in Refactoring Python. `http://blogs.warwick.ac.uk/refactoring/`, Sep. 24, 2006.

[16] Roberts, D., Brant, J. and Johnson, R. A Refactoring Tool for Smalltalk. Theory and Practice of Object Systems (TAPOS), special issue on software reengineering, 3(4):253–263, 1997.

[17] Szabó-Nacsa, R. and Diviánszky, P. and Horváth, Z. Prototype Environment for Refactoring Clean Programs. In *The Fourth Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, July 1–4*, 2004.