Cleaning up Erlang Code is a Dirty Job but Somebody's Gotta Do It

Thanassis Avgerinos
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
ethan@softlab.ntua.gr

Konstantinos Sagonas
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
kostis@cs.ntua.gr

Abstract

This paper describes opportunities for automatically modernizing Erlang applications, cleaning them up, eliminating certain bad smells from their code and occasionally also improving their performance. In addition, we present concrete examples of code improvements and our experiences from using a software tool with these capabilities, tidier, on Erlang code bases of significant size.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering

General Terms Design, Languages

Keywords program transformation, refactoring, code cleanup, code simplification, Erlang

1. Introduction

Most programmers write code. Good programmers write code that works. Very good programmers besides writing code that works also rewrite their code in order to simplify it, clean it, and make it more succinct, modern and elegant. While there will probably never be any real substitute for very good programmers, one might wonder whether there is some intrinsic reason why certain code rewriting tasks cannot be automated and become part of the development tool suite so that even good programmers can readily and effortlessly employ them on their code.

This question has been bothering us for quite some time now, in Erlang and elsewhere. Rather than just pondering it, we decided to embark on a project aiming to automate the modernization, clean up and simplification of Erlang programs. We started by standing on the shoulders of `erl_tidy`, a module of the `syntax_tools` application of Erlang/OTP written by Richard Carlsson, but as we will soon see we have significantly extended it in functionality, features and user-friendliness. The resulting tool is called tidier.

Tidier is a software tool that modernizes and cleans up Erlang code, eliminates certain bad smells from it, simplifies it and improves its performance. In contrast to other refactoring tools for Erlang, such as RefactorErl [9] and Wrangler [7], tidier is completely automatic and not tied to any particular editor or IDE. Instead, tidier comes with a suite of code refactorings that can be selected by its user via appropriate command-line options and applied in bulk on a set of modules or applications. This paper provides only a bird's eye view of the transformations currently performed by tidier; a complete description of tidier and its capabilities is presented in a companion paper [11]. Instead, the main goal of this paper is to report our experiences from using tidier, shed light on some opportunities for code cleanups on existing Erlang source code out there and raise the awareness of the Erlang community on these issues.

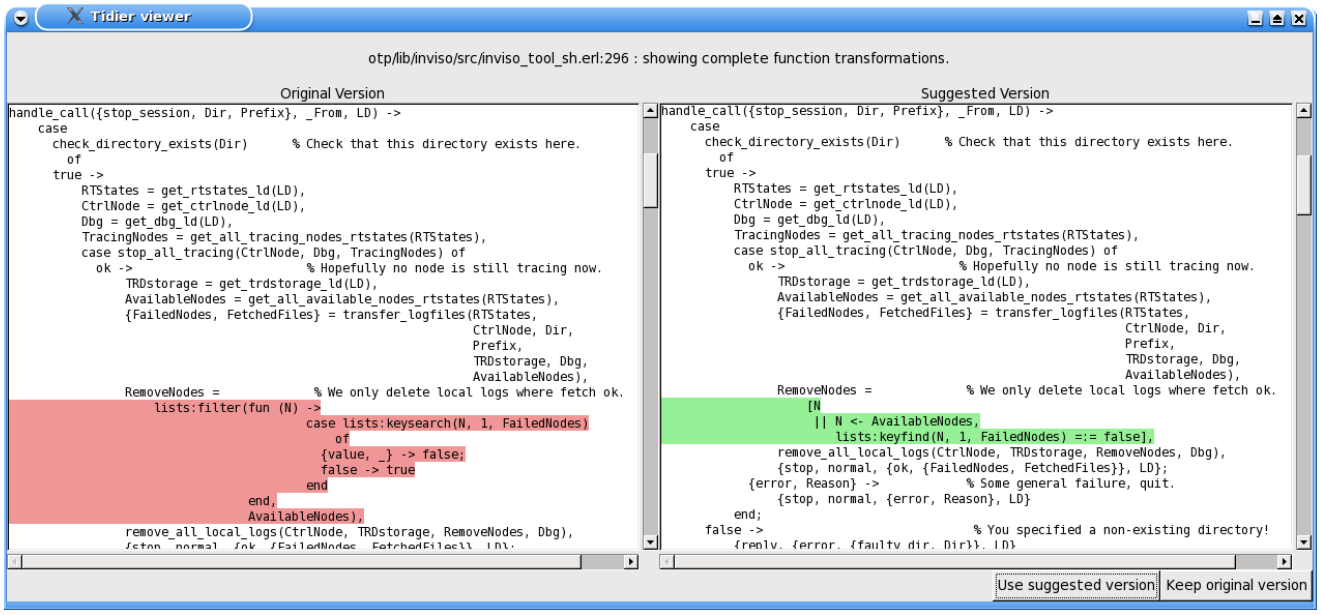
The next section contains a brief presentation of tidier. The main section of this paper, Section 3, gives a captule review for each refactoring currently performed by tidier and shows interesting code fragments we have encountered while trying out tidier on various open source Erlang applications. Section 4 presents tables showing the number of opportunities for tidier's refactorings on several code bases of significant size and discusses tidier's effectiveness. Section 5 presents characteristics of Erlang code that currently prevent tidier from performing more aggressive refactorings, while at the same time preserving its main characteristics, and discusses planned future improvements. The paper ends with some concluding remarks.

Erlang'09, September 5, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-507-9/09/09. . . $5.00

2. Tidier: Characteristics and Overview

From the beginning we set a number of primary goals for tidier:

- Tidier should support a fully automatic mode, meaning that all the refactorings should be such that they can applied on programs without user confirmation.
- Tidier should be flexible. Users should be able to decide about the set of refactorings that they want from tidier and, if they choose so, supervise or even control the refactorings that are performed.
- Tidier should never be wrong. Due to its fully automatic nature, tidier should perform a refactoring only if it is absolutely certain that the transformations performed are semantics-preserving, even if this comes at the cost of missing some opportunity or performing some weaker but safer refactoring.
- Tidier's refactorings should be natural and as good as they get. The resulting code should, up to a certain extent, resemble the code that experienced Erlang programmers would have written if they performed these refactorings by hand.
- Tidier should be easy to use and not be bound to any particular editor or IDE.
- Tidier should be fast. So fast that it can be included in the typical `make` cycle of applications without imposing any significant overhead; ideally, an overhead that is hard to notice.



otp/lib/inviso/src/inviso_tool_sh.erl:296 : showing complete function transformations.

Original Version

```
handle_call({stop_session, Dir, Prefix}, _From, LD) ->
    case
      check_directory_exists(Dir)      % Check that this directory exists here.
        of
      true ->
          RTStates = get_rtstates_ld(LD),
          CtrlNode = get_ctrlnode_ld(LD),
          Dbg = get_dbg_ld(LD),
          TracingNodes = get_all_tracing_nodes_rtstates(RTStates),
          case stop_all_tracing(CtrlNode, Dbg, TracingNodes) of
            ok ->                        % Hopefully no node is still tracing now.
                TRDstorage = get_trdstorage_ld(LD),
                AvailableNodes = get_all_available_nodes_rtstates(RTStates),
                {FailedNodes, FetchedFiles} = transfer_logfiles(RTStates,
                                                                CtrlNode, Dir,
                                                                Prefix,
                                                                TRDstorage, Dbg,
                                                                AvailableNodes),
                RemoveNodes =             % We only delete local logs where fetch ok.
                    lists:filter(fun (N) ->
                                         case lists:keysearch(N, 1, FailedNodes)
                                             of
                                           {value, _} -> false;
                                           false -> true
                                         end
                                 end,
                                 AvailableNodes),
                remove_all_local_logs(CtrlNode, TRDstorage, RemoveNodes, Dbg),
                {stop, normal, {ok, {FailedNodes, FetchedFiles}}, LD};
```

Suggested Version

```
handle_call({stop_session, Dir, Prefix}, _From, LD) ->
    case
      check_directory_exists(Dir)      % Check that this directory exists here.
        of
      true ->
          RTStates = get_rtstates_ld(LD),
          CtrlNode = get_ctrlnode_ld(LD),
          Dbg = get_dbg_ld(LD),
          TracingNodes = get_all_tracing_nodes_rtstates(RTStates),
          case stop_all_tracing(CtrlNode, Dbg, TracingNodes) of
            ok ->                        % Hopefully no node is still tracing now.
                TRDstorage = get_trdstorage_ld(LD),
                AvailableNodes = get_all_available_nodes_rtstates(RTStates),
                {FailedNodes, FetchedFiles} = transfer_logfiles(RTStates,
                                                                CtrlNode, Dir,
                                                                Prefix,
                                                                TRDstorage, Dbg,
                                                                AvailableNodes),
                RemoveNodes =             % We only delete local logs where fetch ok.
                    [N
                     || N <- AvailableNodes,
                        lists:keyfind(N, 1, FailedNodes) =:= false],
                remove_all_local_logs(CtrlNode, TRDstorage, RemoveNodes, Dbg),
                {stop, normal, {ok, {FailedNodes, FetchedFiles}}, LD};
            {error, Reason} ->            % Some general failure, quit.
                {stop, normal, {error, Reason}, LD}
          end;
      false ->                            % You specified a non-existing directory!
          {reply, {error, {faulty_dir, Dir}}, LD}
```

Figure 1. Tidier in action: simplifying the source code of a file from the `inviso` application of Erlang/OTP R13B.

Furthermore, we set a list of criteria that would serve as indicators of whether a specific refactoring should be performed by tidier. The transformations should result in code

modernizations: tidier should remove obsolete language constructs and use the most modern construct for the job.

simplifications: The resulting code should be shorter, simpler and therefore more elegant.

with fewer redundancies: The resulting code should contain fewer redundancies than the original version.

with the same or better performance: The new code should not deteriorate in performance and if possible become even faster.

Modes of using tidier One of our goals has been that tidier should be very easy to use. Indeed, the simplest way to use tidier on some Erlang file is via the command:

```
> tidier myfile.erl
```

If all goes well, this command will automatically refactor the code of `myfile.erl` and overwrite the contents of the file with the resulting source code (also leaving a backup file in the same directory). Multiple source files can also be given. Alternatively, the user can tidy a whole set of applications by a command of the form:

```
> tidier applic1/src ... applicN/src
```

which will tidy all `*.erl` files somewhere under these directories. Both of these commands will apply the default set of transformations on all files. If only some of the transformations are desired, the user can select them via appropriate command-line options. For example, one can issue the command:

```
> tidier --guards --case-simplify myfile.erl
```

to only rewrite guards to their modern counterparts (Section 3.1) and simplify all `case` expressions (Section 3.3) of `myfile.erl`. We refer the reader to tidier's manual for the complete and up-to-date set of command-line options.

A very handy option is the `-n` (or `--no-transform`) option that will cause tidier to just print on the standard output the list of transformations that would be performed on these files, together with their lines, without performing them. Alternatively the user can use the `-g` (or `--gui`) option to invoke tidier's GUI and perform refactoring interactively. We expect that novice tidier users will probably prefer this mode of using tidier, at least initially.

Let us examine tidier's GUI. Figure 1 shows tidier in action. In fact, the snapshot depicts tidier refactoring a file from the `inviso` application of Erlang/OTP R13B. Tidier has identified some code as a candidate for simplification and shows the final version of this code to its user. What the snapshot does not show is that that the simplification involves three different refactorings and that tidier has previously shown all these refactorings, one after the other, to its user. At the point when the snapshot is taken, Tidier's GUI shows the old code (on the left) and the new code (on the right); the code parts that differ between the two versions are coloured appropriately (with red color the old excerpt of the code and with green the new). At this point, the user can either press the "Use suggested version" button to accept tidier's transformation or the "Keep original version" button to bypass it. In either case, tidier will continue with the next refactoring or exit if this is the last one.

As a side comment, at some point during tidier's development we were thinking of giving the user the possibility to edit the code on the right (i.e., allowing the user to fine-tune tidier's refactorings), but we have given up on this idea as it requires dealing with too many issues which are peripheral to the main goals of tidier (e.g., how should tidier continue if the user inputs code which is syntactically erroneous, should there be an "undo" option, etc.). The user can and should better use an editor for such purposes.

3. Transformations Performed by Tidier

Let us now see the transformations that tidier performs.

3.1 Transformations inherited from `erl_tidy`

Some of tidier's transformations were inherited from the `erl_tidy` module of Erlang/OTP's `syntax_tools` application. They are all quite simple but, since they are part of tidier and the basis for our work, we briefly describe them here.

Modernizing guards

For many years now, the Erlang/OTP system has been supporting two sets of type checking functions: old-style (`atom/1`, `binary/1`, `integer/1`, ...) and new-style ones (`is_atom/1`, `is_binary/1`, `is_integer/1`, ...). All this time, the implicit recommendation has been that applications should gradually convert to using new-style guards, but not all applications have done so. Those that have not recently got one more incentive to do so: the compiler of the R13B release of Erlang/OTP stopped being silent about uses of old-style guards and generates warnings by default.

Note that the modernization of guards is both a rather tedious job for programmers and a task that cannot be automated easily. For example, it cannot be performed by a global search and replace without the programmer's full attention or by a simple `sed`-like script that does not understand what is a guard position in Erlang. Consider the following Erlang code which, although artificial and of really poor code quality, is syntactically valid. It is probably not immediately obvious to the human eye where the guard is.

```
-module(where_is_the_integer_guard).
-export([obfuscated_integer_test/1]).

obfuscated_integer_test(X) ->
    integer(X) =:= integer.

integer(X) when (X =:= infinity);
integer(X) -> integer;
integer(_) -> not_an_integer.
```

In contrast, for an automated refactoring tool like tidier, which understands Erlang syntax, the modernization of guards is a simple and straightforward task.

Eliminating explicit imports

This transformation eliminates all import statements and rewrites all calls to explicitly imported functions as remote calls as shown:

```
-import(m1, [foo/1]).
-import(m2, [bar/2]).

t(X) ->
   case foo(X) of
   ...
      bar(A, B),
   ...
```

$\Longrightarrow$

```
t(X) ->
   case m1:foo(X) of
   ...
     m2:bar(A, B),
   ...
```

Admittedly, to a large extent the *eliminating imports* refactoring is a matter of taste. Its primary goal is not to make the code shorter but to improve its readability and understandability by making clear to the eye which calls are calls to module-local functions and which are remote calls. In addition, in large code bases, it makes easier to find (e.g. using tools like Unix's `grep`) all calls to a specific `m:f` function of interest. Of course, it is possible to do the above even in files with explicit imports, but it is often more difficult.

Eliminating appends and subtracts

This is a very simple refactoring that substitutes all occurrences of calls to `lists:append/2` and `lists:subtract/2` functions with the much shorter equivalent operators `++` and `--`. The main purpose of this refactoring is to reduce the size of source code but also improve readability.

Transforming maps and filters to list comprehensions

This is a modernization refactoring that involves the transformation of `lists:map/2` and `lists:filter/2` to an equivalent list comprehension. The goals of this transformation are threefold: (a) reduce the source code size; (b) express the mapping or filtering of a list in a more elegant way and (c) increase the opportunities for further refactorings that involve list comprehensions as we will see.

Transforming fun expressions to functions

This is the Erlang analogue of the *extract method* refactoring in object oriented languages [5]. This particular refactoring removes `fun` expressions from functions and transforms them into module local functions. This transformation primarily aims at improving code readability but can also be used for detecting opportunities for clone removal as also noted by the developers of Wrangler [6].

3.2 Simple transformations

From this point on, all transformations we present are not performed by `erl_tidy`. We start with the simple ones.

Transforming coercing to exact equalities and inequations

In the beginning, the Erlang Creator was of the opinion that the only reasonable numbers were arbitrary precision integers and consequently one equality (`==`) and one inequation (`/=`) symbol were sufficient for comparing between different numbers. At a later point, it was realized that some programming tasks occasionally also need to manipulate floating point numbers and consequently Erlang was enriched by them. Most probably, because C programmers were accustomed to `==` having coercing semantics for numbers, comparison operators for exact equality (`=:=`) and inequation (`=/=`) were added to the language. These operators perform *matching* between numbers. Up to this point all is fine. The problem is that in 99% of all numeric comparisons, Erlang programmers want matching semantics but use the coercing equality and inequation operators instead, probably unaware of the distinction between them or its consequences for readability of their programs by others.

Tidier employs local type analysis to find opportunities for transforming coercing equalities and inequation with an integer to their matching counterparts. The analysis, although conservative, is often quite effective. The transformation itself is trivial.

Modernizing functions

As the Erlang language and its implementation evolve, some library functions become obsolete. These functions typically get replaced by some other function with similar functionality. Occasionally a new function which is cleaner and/or more efficient than the old one is added in the library and recommended as their replacement.

As a rather recent such example, we discuss in detail the case of the commonly used library function `lists:keysearch/3`. This function returns either a pair of the form {`value,Tuple`} or the atom `false`. Throughout the years, it was repeatedly noticed by various Erlang programmers that `Tuple` is a tuple, a whole tuple and nothing but a tuple, so wrapping it in another tuple in order to distinguish it from the atom `false` is completely unnecessary. As a result, R13 introduced the library function `lists:keyfind/3` which has the functionality of `lists:keysearch/3` but instead returns either `Tuple` or `false`. Notice that a simple function renaming refactoring and removing the `value` wrapper do *not* suffice in this case. To see this, consider the following excerpt from the code of Erlang/OTP R13B's `lib/stdlib/src/supervisor.erl:800`:

```
case lists:keysearch(Child#child.name, Pos, Res) of
  {value, _} -> {duplicate_child, Child#child.name};
  _ -> check_startspec(T, [Child|Res])
end
```

To preserve the semantics, this code should be changed to:

```
case lists:keyfind(Child#child.name, Pos, Res) of
  false -> check_startspec(T, [Child|Res]);
  _ -> {duplicate_child, Child#child.name}
end
```

and indeed this is the transformation that tidier performs, based on type information about the return values of the two functions. Moreover, notice that there are calls to `lists:keysearch/3` that *cannot* be changed to `lists:keyfind/3`. One of them, where the matching is used as an assertion, is shown below:

```
...
{value, _} = lists:keysearch(delete, 1, Query),
...
```

This particular transformation involving `lists:keysearch/3` is just one member of a wider set of similar function modernizations that are currently performed by tidier. Their purpose is to assist programmers with software maintenance and upgrades. Judging from the number of obsolete function warnings we have witnessed remaining unchanged across different releases, both in Erlang/OTP and elsewhere, it seems that in practice updating deprecated functions is a very tedious task for Erlang programmers to perform manually.

Record transformations

The *record transformations* refactoring refers to a series of record-related transformations that are performed by tidier. Detailed examples can be found in the companion paper [11] but briefly the refactoring consists of three main transformation steps: (i) converting `is_record/[2,3]` guards to clause matchings; (ii) generating fresh variables for the record fields that are used in the clause and matching them with the corresponding fields in the clause pattern; (iii) replacing record accesses in the clause body with the new field variables. Record transformations lead to shorter and cleaner code, improve code readability, may trigger further refactorings, and when applied *en masse* they can even improve performance.

3.3 Transformations that eliminate redundancy

Various refactorings specialize the code and remove redundancies.

Specializing the `size/1` function

Tidier employs this refactoring to find opportunities to specialize the `size/1` function. Since Erlang/OTP R12 there exist two new BIFs that return the size of tuples (`tuple_size/1`) and binaries (`byte_size/1`). By performing a local type analysis, tidier automatically performs this substitution whenever possible. Such a refactoring has a lot of benefits: (i) modernizes the code; (ii) makes the programmers' intentions about types clear rather than implicit; (iii) assists bug detection tools like Dialyzer [8] to detect type clashes with less effort; (iv) slightly improves the performance of programs; and (v) often triggers further simplifications.

Simplifying guard sequences

This refactoring removes redundant guards and simplifies guard sequences. Some examples are shown below (the `when` is not shown) where we have taken the liberty to combine guard simplifications with some other refactorings we have previously introduced.

```
is_list(L), length(L) > 42    ⟹  length(L) > 42
is_integer(N), N == 42        ⟹  N =:= 42
is_tuple(L), size(T) < 42     ⟹  tuple_size(T) < 42
```

Such refactorings reduce the code size (both source and object) and also improve performance.

Structure reuse

The *structure reuse* refactoring is quite similar to (and inspired from) transformations that optimizing compilers perform. Identical structures (tuples or lists) in the same clause containing fully evaluated terms (i.e., not calls) as subterms are identified by tidier and their first occurrences are assigned to fresh variables. When the identification phase is over, tidier simply replaces all subsequent occurrences of the identical structures with the new variables. This refactoring reduces the code size and also improves performance.

Straightening case expressions

We use the term *straightening* to describe the refactoring of a `case` expression to a matching statement. Such a refactoring can only be applied when the `case` expression has only one alternative clause. Tidier identifies those cases and performs this transformation provided that the body of the `case` does not contain any comments (presumably commented-out alternative `case` clauses or some message that the treatment in the `case` body is currently incomplete).

Temporary variable elimination

This is another refactoring inspired from compiler optimizations, namely from copy propagation. Temporarily storing an intermediate result in a variable to be used in the immediately following expression is actually commonplace in almost all programming languages. Tidier, by performing this refactoring, eliminates the temporary variable and replaces it with its value. This transformation, combined with the *straightening* refactoring of the previous paragraph can lead to significant simplifications. For example, consider the following fragment from the development version of Ejabberd's source code (file `src/ejabberd_c2s.erl:1951`, with one variable renamed so that the code fits here):

```
get_statustag(P) ->
  case xml:get_path_s(P, [{elem, "status"}, cdata]) of
    ShowTag -> ShowTag
  end.
```

by straightening the `case` expression and eliminating the temporary variable the code will be transformed by tidier to:

```
get_statustag(P) ->
  xml:get_path_s(P, [{elem, "status"}, cdata]).
```

However, if tidier applied this refactoring aggressively, we would end up with code 'simplifications' that would look completely unnatural and most probably would never be performed by a programmer. An example of unwanted behaviour from this refactoring is illustrated below:

```
get_results(BitStr) ->
  Tokens = get_tokens(BitStr),
  ServerInfo = get_server_info(Tokens),
  process_data(ServerInfo).
```

$\not\Downarrow$

```
get_results(BitStr) ->
  process_data(get_server_info(get_tokens(BitStr))).
```

Since only few Erlang programmers would consider the resulting code an improvement over the original one as far as code readability is concerned, tidier does *not* perform such refactorings.

Instead, tidier performs the *temporary variable elimination* refactoring when:

- The variable that was used to store the temporary result is eventually used to return the result of a clause (as in the first example we saw).
- It is determined that such a refactoring can lead to further and more radical refactorings later on (such as the ones we will present in Section 3.4). In this case, to ensure that such refactorings are possible after the transformation, tidier has to perform a speculative analysis about the result of further refactorings after this transformation.

Simplifying expressions

While reviewing Erlang code fragments, we have come across a conglomeration of expression simplifications that could be achieved just by applying some simple transformations. Specifically, a very frequent case involved the simplification of boolean `case` and `if` expressions.

As an actual such example, the first transformation of Figure 2 shows the simplification of source code from Erlang Web (file `wparts-1.2.1/src/wtype_time.erl:177`). In this case the code will be simplified further by tidier when the `is_between/3` guard Erlang Enhancement Proposal [10] is accepted and by unfolding the `lists:all/2` call as shown in the second transformation of the same figure. This last step is not done yet.

```
is_valid_time({H1, H2, H3}) ->
  Hour = if (H1 >= 0) and (H1 < 24) -> true;
            true -> false
         end,
  Minute = if (H2 >= 0) and (H2 < 60) -> true;
              true -> false
           end,
  Sec = if (H3 >= 0) and (H3 < 60) -> true;
           true -> false
        end,
  lists:all(fun(X) -> X == true end,
            [Hour, Minute, Sec]).
```

⇓

```
is_valid_time({H1, H2, H3}) ->
  Hour = (H1 >= 0) and (H1 < 24),
  Minute = (H2 >= 0) and (H2 < 60),
  Sec = (H3 >= 0) and (H3 < 60),
  lists:all(fun (X) -> X == true end,
            [Hour, Minute, Sec]).
```

⋮

⇓

```
is_valid_time({H1, H2, H3}) ->
  Hour = is_between(H1, 0, 23),
  Minute = is_between(H2, 0, 59),
  Sec = is_between(H3, 0, 59),
  Hour andalso Minute andalso Sec.
```

Figure 2. A case of multiple `if` simplifications.

3.4 Simplification of list comprehensions

Although the list comprehension transformations that were inherited from `erl_tidy` are semantically correct, at times, the resulting code was not what an expert Erlang programmer would have written if she were transforming the code by hand. The refactorings in this section describe a series of transformations that are supported by tidier in order to improve the quality of the list comprehensions that are produced and at the same time simplify them even more by using the refactorings that were presented in the previous sections.

Fun to direct call

This is a very simple transformation. It is typically performed in conjunction with the refactoring that transforms a `fun` expression to a local function (Section 3.1), and transforms the application of a function variable to some arguments to a direct call to the local function with the same argument.

Inlining simple and boolean filtering funs

A simple `fun` within a `lists:map/2` or `lists:filter/2` which is defined by a match all clause without guards can be inlined when the `map` or `filter` call is transformed to a list comprehension. This simplifies the resulting code and simultaneously makes it more appealing and natural to the programmer's eye. We illustrate it:

```
lists:filter(fun (X) -> is_gazonk(X) end, L)
```

⇓

```
[X || X <- L, is_gazonk(X)]
```

One more case where it is possible to do a transformation similar to the above is when the `fun` is used in `lists:filter/2` and defines a *total* boolean function (i.e., a function that does not impose any constraints on its argument) as the code below (from Erlang/OTP R13B's `lib/kernel/src/pg2.erl:278`):

```
del_node_members([[Name, Pids] | T], Node) ->
  NewMembers =
    lists:filter(fun(Pid) when node(Pid) =:= Node -> false;
                    (_) -> true
                 end, Pids),
  ...
```

which tidier automatically transforms to:

```
del_node_members([[Name, Pids] | T], Node) ->
  NewMembers = [Pid || Pid <- Pids, node(Pid) =/= Node],
  ...
```

Deforestation in map+filter combinations

Some nested calls to `lists:map/2` and `lists:filter/2` are transformed to a single list comprehension by tidier, thus eliminating the intermediate list and effectively performing *deforestation* [13] at the source code level. (The companion paper [11] contains an interesting such example.) Whenever the calls to `map` and `filter` are not nested, tidier performs a speculative analysis employing the *temporary result elimination* refactoring from Section 3.3 to see if this can create further opportunities for deforestation. In either case, tidier will perform the deforestation *only* in cases it is certain that doing so will not alter the exception behaviour of the code (e.g., miss some exception that the original code generates). We will come back to this point in Section 5.

Zipping and unzipping

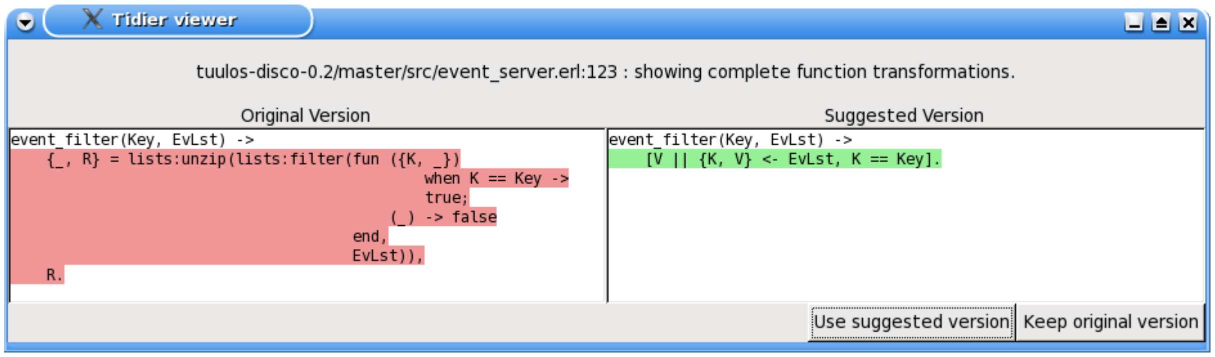
In general, type information (hard-coded or automatically inferred through analysis) can radically improve the resulting refactorings. For example, tidier has hard-coded information that the result of `lists:zip/2` is a list of pairs. This allows tidier to perform function inlining in cases that it would not have been possible without such information. It also prepares tidier for the possibility that *comprehension multigenerators* become part of the language.

Since tidier is treating calls to `lists:zip/2` specially, it felt natural that calls to `lists:unzip/1` would also receive special treatment. One very interesting case appears in the source code of `disco-0.2/master/src/event_server.erl:123`. We show tidier performing a non-trivial code transformation including this refactoring in Figure 3.

3.5 Transformations that reduce the complexity of programs

One of the blessings of high-level languages such as Erlang is that they allow programmers to write code for certain programming tasks with extreme ease. Unfortunately, this blessing occasionally turns into a curse: programmers with similar ease can also write code using a language construct that has the wrong complexity for the task.

Perhaps the most common demonstration of this phenomenon is unnecessarily using the `length/1` built-in function as a test. While



tuulos-disco-0.2/master/src/event_server.erl:123 : showing complete function transformations.

Original Version

```
event_filter(Key, EvLst) ->
    {_, R} = lists:unzip(lists:filter(fun ({K, _})
                                              when K == Key ->
                                              true;
                                          (_) -> false
                                      end,
                                      EvLst)),
    R.
```

Suggested Version

```
event_filter(Key, EvLst) ->
    [V || {K, V} <- EvLst, K == Key].
```

Figure 3. Tidier simplifying the code of `disco-0.2/master/src/event_server.erl`.

this is something we have witnessed functional programming novices do also in other functional languages (e.g., in ML), the situation is more acute in Erlang because Erlang allows `length/1` to also be used as a guard. While most other guards in Erlang have a constant cost and are relatively cheap to use, the cost of `length/1` is proportional to the size of its argument. Erlang programmers sometimes write code which gives the impression that they are totally ignorant of this fact.

Consider the following code excerpt from Erlang/OTP R13B's `lib/xmerl/src/xmerl_validate.erl:542`:

```
star(_Rule,XML,_,_WSa,Tree,_S) when length(XML) =:= 0 ->
  {[Tree],[]};
star(Rule,XMLs,Rules,WSaction,Tree,S) ->
  ... % recursive case of star function here ...
    star(Rule,XMLs2,Rules,WSaction,Tree++WS++[Tree1],S)
  end.
```

The use of `length/1` to check whether a list is empty is totally unnecessary; tidier will detect this and transform this code to:

```
star(_Rule,[],_,_WSa,Tree,_S) ->
  {[Tree],[]};
star(Rule,XMLs,Rules,WSaction,Tree,S) ->
  ... % recursive case of star function here ...
    star(Rule,XMLs2,Rules,WSaction,Tree++WS++[Tree1],S)
  end.
```

thereby changing the complexity of this function from quadratic to linear.

The above is not a singularity. Tidier has discovered plenty of Erlang programs which use `length` to check whether a list is empty. Occasionally some programs are not satisfied with traversing just one list to check if it is empty but traverse even more, as in the code excerpt in Figure 4. Tidier will automatically transform the two `length/1` guards to exact equalities with the empty list (e.g., `AllowedNodes =:= []`). Note that this transformation is safe to do because the two `lists:filter/2` calls which produce these lists supply tidier with enough information that the two lists will be proper and therefore the guards will not fail due to throwing some exception.

Tidier has also located a clause with three unnecessary calls to `length/1` next to each other. The code is from the latest released version of RefactorErl. Its refactoring is shown in Figure 5. Neither we nor tidier understand the comment in Hungarian, but we are pretty sure that the whole `case` statement can be written more simply as:

```
  SideEffs =/= [] orelse UnKnown =/= []
                  orelse DirtyFunc =/= []
```

```
choose_node({PrefNode, TaskBlackNodes}) ->
  ...
    % ..and choose the ones that are not 100% busy.
    AvailableNodes = lists:filter(fun({Node, _Load}) ->
                                          ...
                                  end, AllNodes),
    AllowedNodes = lists:filter(fun({Node, _Load}) ->
                                        ...
                                end, AvailableNodes),
    if length(AvailableNodes) == 0 -> busy;
       length(AllowedNodes) == 0 ->
         {all_bad, length(TaskNodes), length(AllNodes)};
       true ->
         % Pick the node with the lowest load.
         [{Node, _}|_] = lists:keysort(2, AllowedNodes),
         Node
    end;
  ...
```

Figure 4. Code with two unnecessary calls to `length/1` (from the code of `disco-0.2/master/src/disco_server.erl:280`).

thereby saving five lines of code (eight if one also includes the comments) and also avoiding the unnecessary tuple construction and deconstruction.

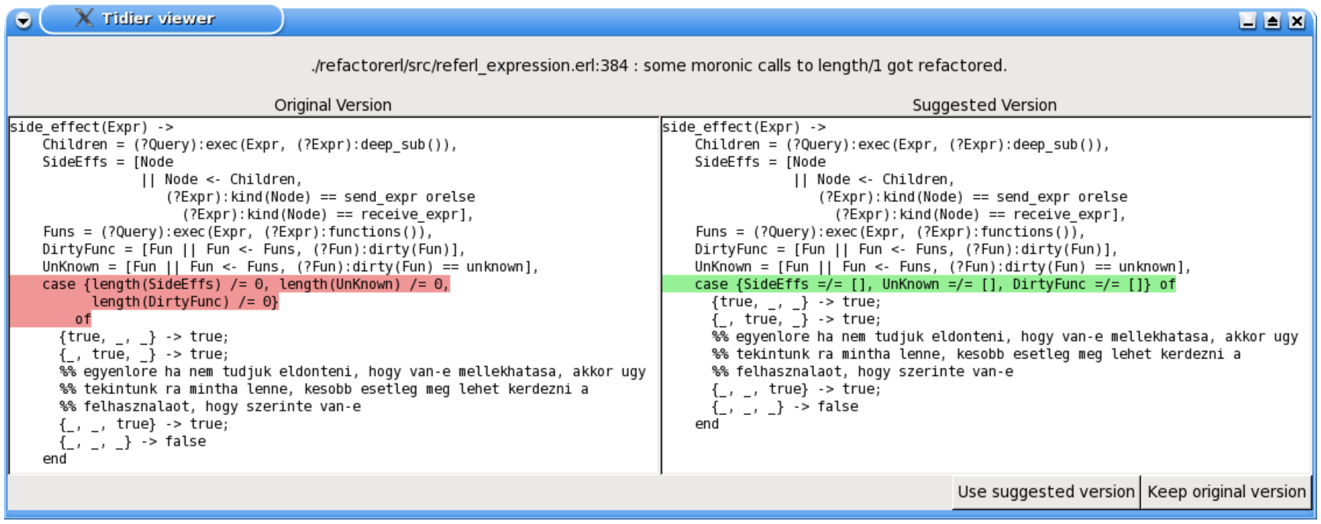
Similar cases also exist which check whether a list contains just one or more that one elements (e.g., `length(L) > 1`). Whenever relatively easy to do, tidier transforms them as in the case shown below (from the code of `lib/ssl/src/ssl_server.erl:1139`) where tidier has also eliminated the call to `hd/1` as part of the transformation.

```
decode_msg(<<_, Bin/binary>>, Format) ->
  Dec = ssl_server:dec(Format, Bin),
  if length(Dec) == 1 -> hd(Dec);
     true -> list_to_tuple(Dec)
  end.
```

$$\Downarrow$$

```
decode_msg(<<_, Bin/binary>>, Format) ->
  Dec = ssl_server:dec(Format, Bin),
  case Dec of
    [Dec1] -> Dec1;
    _ -> list_to_tuple(Dec)
  end.
```

In some other cases though, the code also contains other guard checks which complicate the transformation. For example, consider function `splice/1` from the source code of ErlIDE (located in file `org.erlide.core/erl/pprint/erlide_pperl.erl:171`):



./refactorerl/src/referl_expression.erl:384 : some moronic calls to length/1 got refactored.

Original Version

```
side_effect(Expr) ->
    Children = (?Query):exec(Expr, (?Expr):deep_sub()),
    SideEffs = [Node
                || Node <- Children,
                   (?Expr):kind(Node) == send_expr orelse
                     (?Expr):kind(Node) == receive_expr],
    Funs = (?Query):exec(Expr, (?Expr):functions()),
    DirtyFunc = [Fun || Fun <- Funs, (?Fun):dirty(Fun)],
    UnKnown = [Fun || Fun <- Funs, (?Fun):dirty(Fun) == unknown],
    case {length(SideEffs) /= 0, length(UnKnown) /= 0,
          length(DirtyFunc) /= 0}
        of
      {true, _, _} -> true;
      {_, true, _} -> true;
      %% egyenlore ha nem tudjuk eldonteni, hogy van-e mellekhatasa, akkor ugy
      %% tekintunk ra mintha lenne, kesobb esetleg meg lehet kerdezni a
      %% felhasznalaot, hogy szerinte van-e
      {_, _, true} -> true;
      {_, _, _} -> false
    end
```

Suggested Version

```
side_effect(Expr) ->
    Children = (?Query):exec(Expr, (?Expr):deep_sub()),
    SideEffs = [Node
                || Node <- Children,
                   (?Expr):kind(Node) == send_expr orelse
                     (?Expr):kind(Node) == receive_expr],
    Funs = (?Query):exec(Expr, (?Expr):functions()),
    DirtyFunc = [Fun || Fun <- Funs, (?Fun):dirty(Fun)],
    UnKnown = [Fun || Fun <- Funs, (?Fun):dirty(Fun) == unknown],
    case {SideEffs =/= [], UnKnown =/= [], DirtyFunc =/= []} of
      {true, _, _} -> true;
      {_, true, _} -> true;
      %% egyenlore ha nem tudjuk eldonteni, hogy van-e mellekhatasa, akkor ugy
      %% tekintunk ra mintha lenne, kesobb esetleg meg lehet kerdezni a
      %% felhasznalaot, hogy szerinte van-e
      {_, _, true} -> true;
      {_, _, _} -> false
    end
```

Figure 5. Tidier simplifying the code of `refactorerl-0.6/lib/refactorerl/src/referl_expression.erl`.

```
splice(L) ->
  Res = splice(L, [], []),
  case (length(Res) == 1) and is_list(hd(Res)) of
    true -> no;
    _ -> {yes, Res}
  end.
```

Automatically transforming such code to something like the following is future work:

```
splice(L) ->
  Res = splice(L, [], []),
  case Res of
    [Res1] when is_list(Res1) -> no;
    _ -> {yes, Res}
  end.
```

We intend to enhance tidier with more refactorings that detect programming idioms with wrong complexity for the task and improve programs in similar ways.

4. Effectiveness Across Applications

We have applied tidier to a considerable corpus of Erlang programs both in order to ensure that our tool can gracefully handle most Erlang code out there and in order to test its effectiveness. In this section we report our experiences and the number of opportunities for code cleanups detected by tidier on the code of the following open source projects:[1]

Erlang/OTP This system needs no introduction. We just mention that we report results on the source code of R13B totalling about 1,240,000 lines of Erlang code. Many of its applications under `lib` (e.g., `hipe`, `dialyzer`, `typer`, `stdlib`, `kernel`, `compiler`, `edoc`, and `syntax_tools`) had already been fully or partially cleaned up by tidier. Consequently, the number of opportunities for cleanups would have been even higher if such cleanups had not already taken place.

Apache CouchDB is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API [1]. The CouchDB distribution contains ibrowse and mochiweb as components. We used release 0.9.0 which contains about 20,500 lines of Erlang code.

Disco is an implementation of the Map/Reduce framework for distributed computing [2]. We used version 0.2 of Disco. Its core is written in Erlang and consists of about 2,500 lines of code.

Ejabberd is a Jabber/XMPP instant messaging server that allows two or more people to communicate and collaborate in real-time based on typed text [3]. We used the development version of ejabberd from the public SVN repository of the project (revision 2074) consisting of about 55,000 lines of Erlang code.

Erlang Web is an open source framework for applications based on HTTP protocols [4]. Erlang Web supports both `inets` and `yaws` webservers. The source of Erlang Web (version 1.3) is about 10,000 lines of code.

RefactorErl is an refactoring tool that supports the semi-automatic refactoring of Erlang programs [9]. We used the latest release of RefactorErl (version 0.6). Its code base consists of about 24,000 lines of code.

Scalaris is a scalable, transactional, distributed key-value store which can be used for building scalable Web 2.0 services [12]. We used the development version of scalaris from the public SVN repository of the project (revision 278) consisting of about 35,000 lines of Erlang code. This includes the `contrib` directory of scalaris where the source code of Yaws [15] is also included as a component.

Wings 3D is a subdivision modeler for three-dimensional objects [14]. We used the development version of wings from the public SVN repository of the project (revision 608) consisting of about 112,000 lines of Erlang code. This includes its `contrib` directory.

Wrangler is a refactoring tool that supports the semi-automatic interactive refactoring of Erlang programs [7] within emacs or erlIDE, the Erlang plugin for Eclipse. We used the development version of Wrangler from the public SVN repository of the project (revision 678) consisting of about 42,000 lines of Erlang code.

[1] Throughout its development, we have also applied tidier to its own source code but, since we have been performing the cleanups which tidier were suggesting eagerly, we cannot include tidier in the measurements.

	lines of code	new guards	exact numeric equality	`lists:keysearch/3`	record matches	record accesses	`size`	simplifying guards	structure reuse	straighten + `case` simplify	`map` to comprehension	`filter` to comprehension	deforestations	`zip` + `unzip`	`length`
Erlang/OTP	1,240,000	2911	68	751	1805	2168	487	36	1467	77	564	115	4		12
CouchDB	20,500	22	9	8	6	27	31	2	88	3	38			1	
Disco	2,500	11	2	12		2	9		14		11	5		1	2
Ejabberd	55,000	2		78	18	26	6		70	11	134	40	2		
Erlang Web	10,000	7	11	37	1	12	1	1	15	6	35	7	1		2
RefactorErl	24,000		11	3		8			54	1	39	7		3	7
Scalaris	35,000			2	6	6			22		39	22	3		
Wings 3D	112,000	10	13	45	1	24	26		166	11	25	10			
Wrangler	42,000	6	28	141			1	1	110	7	236	47	5	14	2

Table 1. Number of tidier's transformations on various Erlang source code bases.

For all projects with SVN repositories the revisions we mention correspond to the most recent revision on the 12th of May 2009.

The number of opportunities for tidier's transformations on these code bases is shown on Table 1. From these numbers alone, it should be obvious that detecting, let alone actually performing, all these refactorings manually is an extremely strenuous and possibly also error-prone activity. Tidier, even if employed only as a detector of bad code smells, is worth the effort of typing its name on the command line.

Naturally, the number of opportunities for refactorings that tidier recognizes depends on two parameters: size and programming style of a project's code. As expected, the number of refactoring opportunities on the Erlang/OTP system is much bigger in absolute terms than on all the other code bases combined. This is probably due to the size of the code base and probably also due to the fact that some applications of Erlang/OTP were developed by many different programmers, often Erlang old-timers, over a period of years. But we can also notice that it's not only code size that matters. The table also shows smaller code bases offering more opportunities for refactoring than code bases of bigger size.

What Table 1 does not show is tidier's effectiveness. For some columns of the table (e.g., new guards, record matches) tidier's effectiveness is 100% by construction, meaning that tidier will detect all opportunities for these refactorings and perform them if requested to do so. For some other columns of the table (e.g., `lists:keysearch/3`, `map` and `filter` to list comprehension, structure reuse, `case` simplify) tidier can detect all opportunities for these refactorings but might not perform them based on heuristics which try to guess the intentions of programmers or take aesthetic aspects of code into account. For some refactorings, especially those for which type information is required, tidier's effectiveness is currently not as good as we would want it to be. (We will come back to this point in the next section.)

Table 2 contains numbers and percentages of numeric comparisons with `==` and `/=` that are transformed to their exact counterparts and numbers and percentages of calls to `size/1` that get transformed to `byte_size/1` or `tuple_size/1`. As can be seen, tidier's current analysis is pretty effective in detecting opportunities of transforming calls to `size/1` but quite ineffective when it comes to detecting opportunities for transforming coercing equalities and inequations. A global type analysis would definitely improve the situation in this case. (However, bear in mind that achieving 100% on all programs is impossible since there are uses of `==/2` or `size/1` that cannot be transformed to something else, even if tidier were guided by an oracle.)

	exact num. eq.	`size`
Erlang/OTP	68 / 577 = 12%	487 / 645 = 75%
CouchDB	9 / 15 = 60%	31 / 64 = 48%
Disco	2 / 11 = 18%	9 / 9 = 100%
Ejabberd		6 / 11 = 55%
Erlang Web	11 / 15 = 73%	1 / 1 = 100%
RefactorErl	11 / 35 = 31%	
Scalaris		5 / 6 = 83%
Wings 3D	13 / 46 = 28%	
Wrangler	28 / 54 = 52%	1 / 1 = 100%

Table 2. Effectiveness of tidier' refactorings requiring type info.

5. Conservatism of Refactorings

Despite the significant number of refactorings that tidier performs on existing code bases, we stress again that tidier is currently ultra conservative and careful to respect the operational semantics of Erlang. In particular, tidier will never miss an exception that programs may generate, whether deliberately or not.

To understand the exact consequences of this, we show a case from the code of `lib/edoc/src/otpsgml_layout.erl:148` from Erlang/OTP R13B. The code on that line reads:

```
Functions = [E || E <- get_content(functions, Es)],
```

Although to a human reader it is pretty clear that this code is totally redundant and the result of sloppy code evolution from similar code (actually from the code of `lib/edoc/src/edoc_layout.erl`), tidier *cannot* simplify this code to:

```
Functions = get_content(functions, Es),
```

because this transformation will shut off an exception in case function `get_content/2` returns something other than a proper list. To do this transformation, type information about the result of `get_content/2` is required. Currently, tidier is guided only by a function-local type analysis. Extending this analysis to the module level is future work.

Type information can also come in very handy in rewriting calls to `lists:map/2` and `lists:filter/2` to more succinct list comprehensions. Without type information, tidier performs the following transformation:

```
foo(Ps) -> lists:map(fun ({X,Y}) -> X + Y end, Ps).
```

$\Downarrow$

```
foo(Ps) -> [foo_1(P) || P <- Ps].

foo_1({X,Y}) -> X + Y.
```

and cannot inline the body of the auxiliary function and generate the following code:

```
foo(Ps) -> [X + Y || {X,Y} <- Ps].
```

because this better refactoring requires definite knowledge that `Ps` is a list of pairs. Similar issues exist for refactorings involving `lists:filter/2`. Despite being conservative, tidier is pretty effective. In the code of Erlang/OTP R13B, out of the 679 refactorings of `lists:map/2` and `lists:filter/2` to list comprehensions a bit more than half of them (347) actually use the inlined translation.

We mentioned that tidier currently performs deforestation for combinations of `map` and `filter`. A similar deforestation of `map`+`map` combinations, namely the transformation:

```
L1 = lists:map(fun (X) -> m1:foo(X) end, L0),
L2 = lists:map(fun (X) -> m2:bar(X) end, L1)
```

$\not\Downarrow$

```
L2 = [m2:bar(m1:foo(X)) || X <- L0]
```

as also shown in the arrow is *not* performed by tidier because this requires an analysis which determines that functions `m1:foo/1` and `m2:bar/1` are side-effect free. Again, hooking tidier to such an analysis is future work.

6. Concluding Remarks

This paper described opportunities for automatically modernizing Erlang applications, cleaning them up, eliminating certain bad smells from their code, and occasionally also improving their performance. In addition, we presented concrete examples of code improvements and our experiences from using tidier on code bases of significant size.

As mentioned, tidier is completely automatic as a refactorer but with equal ease can be used as a detector of opportunities for code cleanups and simplifications. Tools that aid software development, such as code refactorers, have their place in all languages, but it appears that higher-level languages such as Erlang are particularly suited for making the cleanup process fully or mostly automatic. We intend to explore this issue more.

Acknowledgements

We thank Richard Carlsson, Björn Gustavsson, and Kenneth Lundin for supportive comments and suggestions for refactorings. We also thank Dan Gudmundsson: without the use of his `wx` application, the user interface of tidier would have taken longer to write and would probably look less aesthetically pleasing.

Finally, we thank all developers of projects mentioned in this paper for publicly releasing their code as open source and giving us plenty of opportunities to find nice examples for our paper.

References

[1] The CouchDB project, 2009. `http://couchdb.apache.org/`.

[2] Disco: Massive data, minimal code (version 0.2), Apr. 2009. `http://discoproject.org/`.

[3] Ejabberd community site: The Erlang Jabber/XMPP daemon, 2009. `http://www.ejabberd.im/`.

[4] Erlang Web, May 2009. `http://www.erlang-web.org/`.

[5] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, Massachusetts, 2001.

[6] H. Li and S. Thompson. Clone detection and removal for Erlang/OTP within a refactoring environment. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 169–177, New York, NY, USA, Jan. 2009. ACM.

[7] H. Li, S. Thompson, G. Orösz, and M. Tóth. Refactoring with Wrangler, updated: Data and process refactorings, and integration with Eclipse. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 61–72, New York, NY, USA, Sept. 2008. ACM.

[8] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In C. Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, Nov. 2004.

[9] L. Lövei, Cs. Hoch, H. Köllő, T. Nagy, A. Nagyné-Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring module structure. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 83–89, New York, NY, USA, Sept. 2008. ACM.

[10] R. A. O'Keefe. Erlang Enhancement Proposal: `is_between/3`, July 2008. `http://www.erlang.org/eeps/eep-0016.html`.

[11] K. Sagonas and T. Avgerinos. Automatic refactoring of Erlang programs. In *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, New York, NY, USA, Sept. 2009. ACM.

[12] T. Schütt, F. Schintke, and A. Reinefeld. Scalaris: Reliable transactional P2P key/value store. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 41–48, New York, NY, USA, Sept. 2008. ACM.

[13] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Comput. Sci.*, 73(2):231–248, 1990.

[14] Wings 3D, 2009. `http://www.wings3d.com/`.

[15] Yaws: Yet another web server, 2009. `http://yaws.hyber.org/`.