Automatic Refactoring of Erlang Programs

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

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

Abstract

This paper describes the design goals and current status of tidier, a software tool that tidies Erlang source code, making it cleaner, simpler, and often also more efficient. In contrast to other refactoring tools, tidier is completely automatic and is not tied to any particular editor or IDE. Instead, tidier comes with a suite of code transformations that can be selected by its user via command-line options and applied in bulk on a set of modules or entire applications using a simple command. Alternatively, users can use tidier's GUI to inspect one by one the transformations that will be performed on their code and manually select only those that they fancy. We have used tidier to clean up various applications of Erlang/OTP and have tested it on many open source Erlang code bases of significant size. We briefly report our experiences and show opportunities for tidier's current set of transformations on existing Erlang code out there. As a by-product, our paper also documents what we believe are good coding practices in Erlang. Last but not least, our paper describes in detail the automatic code cleanup methodology we advocate and a set of refactorings which are general enough to be applied, as is or with only small modifications, to the source code of programs written in Haskell or Clean and possibly even in non-functional languages.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—Restructuring, reverse engineering, and reengineering; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Design, Languages

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

1. Introduction

Writing code that is as clean and simple as possible is desirable but also difficult to do in any language, declarative or not. The ability to achieve this is an acquired skill that requires a lot of experience in writing programs in the language, studying source code of others, having pretty good knowledge of the various alternatives of expressing programming intentions using the constructs of the language, but also having quite a lot of discipline when programming.

PPDP'09, September 7–9, 2009, Coimbra, Portugal.

To help programmers write better code, most languages these days come with websites and books that document good coding practices in the hope that programmers will read and follow them. The programming language Erlang is no exception in this respect. Indeed, both the `www.erlang.org` website and the various books on Erlang contain many useful pieces of advice on how to write better programs. Still, at least judging from some open source and commercial code we have laid our eyes upon, it seems that some of this advice has never been read or, even if it has been, it has been largely neglected by some programmers. Once again, Erlang is by no means the only programming language where one can witness this phenomenon. On the contrary, the situation regarding code quality is most probably worse in some other languages, especially non-declarative ones.

Another reason that often contributes to having lots of code of sub-optimal quality at any particular point in time is that most programming languages evolve. For example, some of the language constructs that Erlang programmers can employ today (e.g., funs, binaries, comprehensions, etc.) result in better and more succinct code than code which could be written using Erlang constructs of ten years ago. Still, even nowadays, it is not uncommon to notice members of the Erlang programming community write or post programs that use old-fashioned language constructs or programs that could be written more succinctly and elegantly in modern Erlang. This, coupled with the fact that there is a lot of Erlang code out there that has been written long ago and since then has not been revised or modernized, does not help much in improving the code quality of Erlang applications or in having code bases that teach best practices to language newcomers.

For a long time now, the first author of this paper, both due to obsession with code cleanliness and with a desire to show new members of his team code with good coding practices only, has been manually performing code cleanups in code bases of projects that he has been involved in. (No doubt he is not the only programmer who has ever done so.) Sooner or later, anybody involved in this practice is bound to notice that some code improvements and modernizations are so simple and standardized that they could be automated quite easily. This is especially true for code improvements obtained by using more modern language constructs. In fact, in the `syntax_tools` application [3], the Erlang/OTP system has a module called `erl_tidy` that can be used from within Erlang to perform a limited set of these refactorings. We have decided to use `erl_tidy` as a starting point for our work but we have significantly modified and extended its capabilities in ways that we will shortly describe.

Another set of code improvements that can be automated relatively easily are those which are identical or very similar to transformations that optimizing compilers perform. Some of these transformations, especially high-level ones designed for functional languages, besides improving the running time of programs, have the nice property that they make code smaller and less complicated.

Such source code is typically cleaner and easier to understand and maintain than long spaghetti code. For this reason, we hold and advocate that it is worthwhile to perform such transformations already at the source level, rather than (only) at the level of the compiler's intermediate code representation. Besides the benefits that this has for source code readability, it also makes programs more portable as they become less dependent on the compiler version which is used or even the language implementation on which they will be compiled and executed.

Rather than continuing performing cleanups by hand, we have decided to create a software tool, called tidier, that performs all the above and eases the cleanup and code simplification of Erlang applications. This has allowed us to apply the tool to code bases of significant size and fine-tune its functionality. As we will soon see, tidier is completely automatic, flexible and very easy to use, and performs a suite of code transformations ranging from very simple to quite sophisticated. Although we expect that tidier will be used as an automatic code refactorer in most projects, tidier can also be used only as an automatic detector of certain bad code smells and let the user be in total control of the cleanup process. After all, not all programmers may find all of tidier's refactorings to their liking.

Perhaps we should also point out that some of the transformations that tidier performs actually *increase* the dependency on language versions (i.e., they require the use of a rather recent Erlang/OTP release) and may not be suitable for applications that have requirements to be able to run in older releases. However, rewriting old idioms that were once necessary due to a more simplistic language implementation into concise and modern code is just as efficient (or better) and makes applications more future safe as older language features often get removed as languages evolve.

Outline of the paper To make the paper relatively self-contained, the next section briefly overviews the Erlang language and the evolution of its implementation. Section 3 presents the design characteristics and main properties of tidier. It is followed by the main section of this paper, Section 4, that describes in detail the code transformations currently performed by the tool. In Section 5 we briefly mention how tidier can be used and report on our experiences from using tidier in various applications of Erlang/OTP and in open source code bases. Finally, Section 6 reviews related work and the paper ends with some concluding remarks.

2. Erlang and Erlang/OTP

Erlang is a concurrency oriented, dynamically typed, strict functional programming language. In Erlang, terms are either variables, simple terms, structured terms, or function closures. Variables always begin with a capital letter or an underscore. Simple terms include atoms, process identifiers, integers and floating point numbers. Structured terms are lists (enclosed in brackets) and tuples (enclosed in braces). Structured terms are constructed explicitly and deconstructed using pattern matching. Pattern matching is also used to select function clauses or different branches of `case` statements; the two forms are equivalent and choosing between them is a matter of taste. The program on Figure 1 shows all the above. It also shows how Erlang code is organized in modules, how the code can contain calls to exported functions of some other module (the call to function `math:sqrt/1` in our example), and how pattern matching is enriched by the presence of flat guards such as type tests and arithmetic comparisons.

The Erlang language is rather small, but it has evolved from an even smaller language which over the years has been enriched with new language constructs [1]. For example, for some years now Erlang supports a notation for function closures (known as *funs* in the Erlang lingo) when older Erlang versions only supported `apply`. Similarly, modern Erlang comes with language constructs

```
-module(example).
-export([factorial/1, nth/1, area/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).

nth(1, [H|_]) -> H;
nth(N, [H|T]) when is_integer(N), N > 1 ->
    nth(N-1, T).

area(Shape) ->
    case Shape of
        {square, Side} when is_number(Side) ->
            Side * Side;
        {circle, Radius} ->
            3.14 * Radius * Radius;  %% well, almost
        {triangle, A, B, C} ->
            S = (A + B + C) / 2,
            math:sqrt(S * (S-A) * (S-B) * (S-C))
    end.
```

Figure 1. An example Erlang program.

to perform pattern matching directly on *binaries* and *bit streams* [5] when older Erlang required a conversion of binaries to lists first. Modern Erlang comes with a notation for *records*, which allows referring to tuple elements by name instead of by position. Using record notation and some appropriate declaration, we could for example write the first `case` clause of the `area/1` function of our example program as follows:

```
#square{side = Side} when is_number(Side) ->
    Side * Side;
```

Over the years Erlang has also adopted various constructs from other programming languages, most notably *list comprehensions*, which are a convenient shorthand for a combination of `map`, `filter` and `append` on lists. For example, the following list comprehension:

```
List = [{1,2.56}, {3.14,4}, some_atom, {5,6}],
[Y*(Y+1) || {X,Y} <- List, is_integer(X), X > 1].
```

will silently filter out the `some_atom` element of the list and produce the list `[42]`. On the other hand, in non-filter expressions, the evaluation of list comprehensions might throw a runtime exception. For example, the list comprehension we just showed would throw an exception if `List` also contained the term `{7,eleven}`.

The main implementation of the language is the Erlang/OTP (Open Telecom Platform) system from Ericsson. At the time of this writing the most recent Erlang/OTP version is R13B (release 13B). Besides libraries containing a large set of built-in functions (BIFs) for the language, the Erlang/OTP system comes with a number of ready-to-use components and design patterns (such as finite state machines, generic servers, supervisors, etc.) providing a set of design principles for developing fault-tolerant Erlang applications. Indeed, using the Erlang/OTP system, a number of commercial and open-source applications have been written over the years, making Erlang both one of the most industrially relevant declarative languages and a language with a significant body of existing source code out there.

One problem with having lots of code is that undoubtedly there is also a wide variation in code quality between different code bases; often even *within* the code base of a single application. We have witnessed this phenomenon in many Erlang code bases we have examined. While some projects adopt or even impose rigorous coding standards, others follow a more relaxed attitude in what code can join their code base. Some applications are quick to adopt

newer language constructs that make their code cleaner and simpler, while other projects never modify or modernize their code if it isn't seriously broken. While the above observations are by no means applicable only to Erlang — or to declarative languages in general — we hold that they are particularly relevant for this type of languages because declarative languages: 1) are often moving-targets and more willing to include higher-level constructs in their definition, and 2) besides giving programmers the opportunity to write cleaner and more succinct programs, they also often make it easier for them to write *less* efficient code than what they would have written in some low-level imperative language or in the declarative language given some other, semantically equivalent, language construct. In this respect, writing good code in a declarative language (and Erlang in particular) is actually more difficult than in a language such as C.

However, declarative languages such as Erlang have one clear advantage compared with lower-level, imperative languages. Because of their relatively clean semantics, they are more suited to high-level, semantics-preserving transformations that can automatically detect and/or cleanup source code from certain old-fashioned or less efficient ways of writing some program. To ease the modernization and code improvement of Erlang applications we have developed tidier, an automatic software refactoring tool whose design goals and current set of capabilities we will describe below.

3. Tidier's Design and Goals

Before we describe in detail the code transformations that the current version of tidier performs, we present the design characteristics and main properties of the tool. In doing so, we also implicitly mention how tidier differs from other refactoring tools for Erlang such as Wrangler [7] or RefactorErl [10].

Main characteristics

The main design characteristics of tidier are that it should be:

***fully automatic*:** In particular tidier should provide a mode of operation where it can be applied in bulk to a set of modules or entire applications without requiring any interaction from its user.

***reliable*:** This characteristic is very much related to the previous one. In a semi-automatic refactoring tool, like Wrangler or RefactorErl, it is probably OK to rely on the programmer to confirm and/or take full responsibility for refactorings that might be unsafe in some, hopefully rare, circumstances. In contrast, tidier, being fully automatic, cannot afford this luxury.

***universal and easy to use*:** This means that tidier should not be tied to any particular editor or integrated development environment (IDE). Particular editors and IDEs, no matter how popular or widespread they may be within a particular language community, always leave out a percentage of users who, for their own reasons, choose some other editor or environment to do their development.

***flexible*:** The refactorings performed by tidier should be selectable by the user. Also, if users want to, they should be allowed to conveniently inspect the result of the refactoring process and filter it and/or influence it according to their desires.

***fast*:** The tool should be fast enough so that in most applications it can become part of the typical `make` cycle without imposing any noticeable overhead to the process.

Needless to mention, tidier achieves all the above.

Transformation properties

Regarding the transformations performed by tidier, they should be:

***semantics preserving*:** In particular, the transformations should faithfully respect the operational semantics of Erlang. As we will soon see, in some cases tidier could possibly perform better refactorings if it had accurate knowledge about types of variables or the programmers' intentions. Due to the dynamic nature of Erlang and tidier being a fully automatic tool, such information is often not available. In such cases, tidier should either not perform a refactoring or perform a weaker one that is guaranteed to be semantics preserving.[1]

***code improving*:** A transformation should be performed only if it improves the code according to some criterion. Relevant criteria used by tidier are: (i) the new code uses a more modern Erlang construct (e.g. one which is more succinct or is not obsoleted and retained only for backwards compatibility); (ii) the new code is shorter and more elegant; (iii) the new code has less redundancy or (iv) the new code executes faster.

***syntactically pleasing and natural*:** In particular, the transformations should result in code which is as close as possible to what expert Erlang programmers would have written if they performed the same transformations by hand. Among other things, this means that the transformed source code should be naturally indented and, whenever possible, use variable and function names that accurately reflect the code from which they originated instead of using artificial names such as `Var_4711`.

In addition, if possible, tidier should try to guess the intentions of programmers but never try to outsmart them.

4. Transformations Performed by Tidier

Let us now examine the transformations that tidier performs and the effect that they have on some source code examples. In doing so, we also discuss aspects of transformations that require extra care or make them tricky to implement.

4.1 Simple transformations

We start by describing the simplest transformations. These transformations (and those of Section 4.3) are also provided by the `erl_tidy` module which we used as a starting point for tidier.

Modernizing guards and calls to old-fashioned functions

For many years now, the Erlang/OTP system has been supporting two sets of type checking functions, often used as guards: old-style (`atom/1`, `binary/1`, `integer/1`, ...) and new-style ones (`is_atom/1`, `is_binary/1`, `is_integer/1`, ...). In addition, many commonly used library functions have changed and continue to change names between releases (e.g., `dict:list_to_dict/1` is now called `dict:from_list/1`, `unix:cmd/1` changed name to `os:cmd/1` for political correctness, the `reserved_word/1` function which used to be in `io_lib` is now located in the `erl_scan` module, etc.). The *modernizing function name* refactoring modernizes the guard names and takes care of such function renaming issues. Occasionally, this refactoring is aided by the *eliminating imports* refactoring which expands `-import` directives and exposes the proper module name of function calls. In doing so, it also eases the job of subsequent transformations.

[1] As we will see, some of tidier's refactorings might change the type of exception that is raised by the code, e.g. from `case clause` to `badmatch`. However, we consider such refactorings semantics preserving because they will never result in code that misses some exception that would have been generated or in code that results in some exception being thrown when the original code would not raise one. Also, note that the issue of not preserving the exception behaviour of a program is not tidier-specific but also present in the other refactoring tools for Erlang.

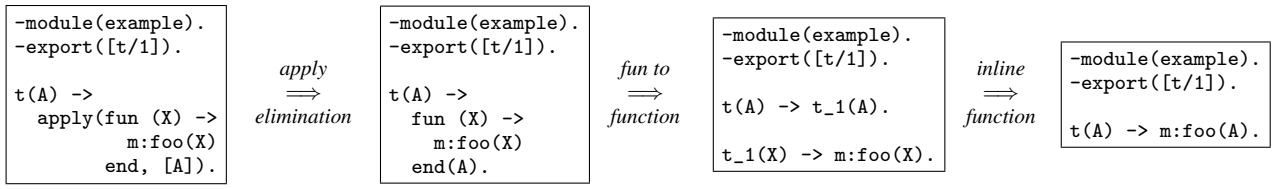

Figure 2. Illustration of refactorings that work hand in hand.

This set of refactorings is pretty straightforward for a software tool that understands Erlang syntax, but quite tedious for programmers and very difficult, if not impossible, to perform with a global search or replace or with a shell script. The interested reader is referred to a companion paper [2] which shows an interesting code example that substantiates this claim.

Turning apply calls to remote calls

Another simple but also very useful transformation is the *apply elimination*. Whenever the last argument of either `apply/2` or `apply/3` is a list whose elements are statically known, the `apply` can be rewritten as a remote function call. For example, the call `apply(M,F,[A1,A2])` can be rewritten as `M:F(A1,A2)`. This refactoring both reduces the code size and improves code readability and understandability. In addition, using remote function calls instead of `apply` may allow other program analysis tools recognize function calls that they would not be able to recognize in the `apply` format. So this refactoring may result in more accurate analyses.

Turning funs into functions

This transformation is known as *lambda lifting* in functional languages and is a special case to what is known as the *extract method* refactoring in the object-oriented refactoring lingo [4]. It transforms a fun expression to local function and changes the point where the fun expression was previously applied to a function call. In order to perform this refactoring, tidier locates all fun expressions in a function clause and replaces them with local function calls. After the function analysis, tidier generates definitions for the local functions that correspond to the replaced fun expressions. These local functions take their name out of the function from which they were extracted extended by a suitable numerical suffix. Occasionally, the newly introduced local functions may have different arities than their corresponding initial fun expressions. This is due to the fact that the fun expression may be using variables that are available within the scope of the source function and therefore have to be passed to the extracted functions.

As an example, Figure 2 shows the effect of these two refactorings and of *inline function* on a very simple module. The reader should notice that even though these refactorings are very simple on their own, their synergy effects considerably simplify the code. This is not something which is restricted to the refactorings of this section; instead, it is a general phenomenon.

4.2 Record transformations

The initial purpose of this refactoring, which is available in tidier but not in `erl_tidy`, was to eliminate uses of `is_record/[2,3]` guards which are somewhat superfluous in modern Erlang. Indeed, in most Erlang programs these guards are not really needed except in cases where they are used in a disjunction to test for different alternatives where at least one of them checks that a term is a record as e.g. in:

```
foo(T) when is_atom(T); is_record(T, rec) ->
  ...
```

With time more transformations were added to that of eliminating `is_record` guards and nowadays we use the name *record transformations* to describe a whole bunch of simple refactorings involving records that tidier performs. We will illustrate them step-by-step using a code example from Erlang/OTP R13B's `lib/ssl/src/ssl_prim.erl:121` (slightly simplified). The initial code fragment is the following:

```
process(St, Pid) when is_record(St, st),
                      St#st.status =:= open,
                      is_pid(Pid) ->
    inet_tcp:controlling_process(St#st.proxysock, Pid).
```

As we can see the variable `St` of the function clause is an `#st{}` record. Tidier will detect this fact and will apply the *record guard to matching* refactoring, which will substitute the `is_record/2` guard with an explicit record matching. The use of a matching instead of a guard is to some extent a matter of taste. However, as we shall soon see, this change can enable further refactorings. The clause after this refactoring becomes:

```
process(#st{} = St, Pid) when St#st.status =:= open,
                              is_pid(Pid) ->
    inet_tcp:controlling_process(St#st.proxysock, Pid).
```

The code is already shorter but this is only the beginning. In the clause body there are two record field accesses (for fields named `status` and `proxysock`). These accesses can be eliminated by introducing fresh variables, using appropriate names, and use a record expression in the clause head to initialize them by pattern matching. Then the record accesses can be replaced by the new variables. After these transformations, the `St` variable is no longer needed and it can also be eliminated. After applying tidier's *record field access elimination* refactoring, the clause becomes:

```
process(#st{status = Status, proxysock = Proxysock}, Pid)
  when Status =:= open, is_pid(Pid) ->
    inet_tcp:controlling_process(Proxysock, Pid).
```

The code can be shortened even more. The newly introduced variable `Status` is only used in an exact equality guard. Therefore this variable can be eliminated and the exact equality test can be replaced by pattern matching. After tidier applies its *equality guard to pattern matching* refactoring, the final form of the code is the one shown below:

```
process(#st{status = open, proxysock = Proxysock}, Pid)
  when is_pid(Pid) ->
    inet_tcp:controlling_process(Proxysock, Pid).
```

How names for fresh variables are chosen Tidier often needs to create fresh variables and give them names. For example, we saw that tidier created variables for the record fields and gave them names which are based on the names of these fields. Actually, this is tidier's second choice. Before generating names for the fresh variables, tidier searches the clause in order to check whether the programmer has already given names to the values of these record

fields (usually via matchings). For example, in a clause like the following:

```
vn(Peer) when is_record(Peer, peer) ->
    MyPid = Peer#peer.pid,
    MyPort = Peer#peer.port,
    {MyPid, MyPort}.
```

the programmer has indicated that the names `MyPid` and `MyPort` are suitable for the `pid` and `port` fields respectively. Thus, tidier will transform this clause to:

```
vn(#peer{pid = MyPid, port = MyPort}) ->
    {MyPid, MyPort}.
```

Whenever none of the above two options are possible for some record field (i.e., there is no user-supplied name for it and the name of the field is already used for some other variable in the clause), tidier will generate a fresh name that is formed by the field name followed by an appropriate integer (e.g., `Port42`).

Experience In the above examples we have demonstrated the result of the *records transformations* on short pieces of code. On large segments of code, the changes are often more extensive and radical. Large code segments may contain multiple record guards or record variables, which tidier handles simultaneously, and may have much more record field accesses, which are often identical in different branches. In our experience, their elimination results in more succinct, better organized, and more readable code. Finally, we also should note that *any* of the refactorings of the *records transformations* can jump start the process. For example, if in our first example the programmer had manually replaced the `is_record/2` guard with a pattern matching, tidier would still have been able to apply the other refactorings, resulting in the same final code.

4.3 Transformations of common list operations

List processing is very common in functional programs and Erlang programs are no exception. It is therefore natural for tidier to pay special effort to simplifying uses of some commonly employed functions of the `lists` module of the Erlang standard library.

Transforming appends and subtracts

The `lists:append/2` and `lists:subtract/2` functions have convenient shorthands, which are also binary operators. This refactoring is trivial and its purpose is to make the source code more succinct. This is illustrated below.

```
...
case lists:append(L1, L2) of
  ...
  L = lists:subtract(L3, [a]),
...
```

$\Longrightarrow$

```
...
case L1 ++ L2 of
  ...
  L = L3 -- [a],
...
```

Transforming maps to comprehensions

The `lists:map/2` function is one of the most frequently used library functions in Erlang. It applies a function to all elements of a list and returns the list with the function's results.

For a number of years now, Erlang has been enhanced with a very powerful and expressive construct, called *list comprehension*, that provides all functionality of a `lists:map/2` and even more. Besides being more powerful, list comprehensions are typically more succinct than maps and arguably also more modern. The *list map to comprehension* refactoring performs the automatic conversion of a `lists:map/2` call to a list comprehension. To perform the actual transformation, tidier introduces a fresh variable in order to create the list generator and applies the function to that variable. Let's see the refactoring on an example:

```
mp(L) ->
    lists:map(fun ({X, Y}) -> X + Y;
                  (X) when is_integer(X) -> 2 * X
              end, L).
```

The semantically equivalent code using a list comprehension is:

```
mp(L) ->
    [fun ({X, Y}) -> X + Y;
         (X) when is_integer(X) -> 2 * X
     end(V) || V <- L].
```

Although more succinct, very few Erlang programmers, if any, would consider the above code an improvement over the original as far as readability is concerned. The situation gets better by tidier automatically applying the *fun to function* (aka *extract method*) refactoring we have discussed in Section 4.1 and also shown in Figure 2. Doing so results in the following code:

```
mp(L) -> [mp_1(V) || V <- L].

mp_1({X, Y}) -> X + Y;
mp_1(X) when is_integer(X) -> 2 * X.
```

which Erlang programmers would most probably find more to their liking. In fact, this is the code that the auto_list_comp option of the `erl_tidy` module would also generate.

Transforming filters to comprehensions

This refactoring is very similar to the previous one and transforms occurrences of `lists:filter/2` to a semantically equivalent list comprehension. Tidier performs this transformation by applying the filtering function to the newly introduced generator variable and places this call as a filter test immediately after the generator. An example of the *list filter to comprehension* refactoring followed by a *fun to function* refactoring is shown below:

```
flt(L) ->
    lists:filter(fun ({X, Y}) -> true;
                     (X) -> is_atom(X)
                 end, L).
```

$\Downarrow$

```
flt(L) ->
    [V || V <- L, fun ({X, Y}) -> true;
                      (X) -> is_atom(X)
                  end(V)].
```

$\Downarrow$

```
flt(L) -> [V || V <- L, flt_1(V)].

flt_1({X, Y}) -> true;
flt_1(X) -> is_atom(X).
```

Once again, the above transformation is quite simple and is also performed by the `erl_tidy` module of Erlang/OTP.

4.4 List comprehension simplifications

Although the *list map/filter to comprehension* refactorings followed by an immediate *fun to function* refactoring results in good looking code we noticed that even better looking code could be generated, especially in certain very commonly occurring cases. We therefore introduced and implemented in tidier the *list comprehension simplifications* refactorings, which describes a family of simple refactorings that can be applied either to list comprehensions which already exist in the code or to those created after applying the *list map/filter to comprehension* refactorings of the previous section. Let us examine these refactorings.

Transforming a fun to a direct call

This is a very simple refactoring that can be applied when the function of the comprehension is just a `fun name/arity` combination (possibly also module-qualified). In this case tidier transforms the fun application to a direct call. The following example illustrates this refactoring on a `lists:map/2` call which exists in the code of Erlang/OTP R13B's `lib/kernel/src/inet_parse.erl:654`.

```
lists:map(fun dig_to_hex/1, lists:reverse(R))
```

⇓

```
[dig_to_hex(V) || V <- lists:reverse(R)]
```

Inlining bodies of simple funs

Whenever the fun definition is simple, the resulting comprehension is not what an expert Erlang programmer would write when transforming the call to `map` or `filter` by hand. In the context of *list comprehension simplifications*, tidier considers a fun definition simple whenever: 1) the fun's argument is a single fresh variable, and 2) the fun's body is either a single call or a boolean expression (for the case of transforming a call to `lists:filter/2` only). In such cases, the fun's body can be inlined in the appropriate place.

We show two examples that illustrate this refactoring. First a case of transforming a call to `lists:map/2`:

```
lists:map(fun (X) -> X + 42 end, L)
```

⇓

```
[X + 42 || X <- L]
```

and also a case of transforming a call to `lists:filter/2`:

```
lists:filter(fun (X) ->
                 is_integer(X) andalso X > 0
             end, L)
```

⇓

```
[X || X <- L, is_integer(X), X > 0]
```

Notice that for preserving the semantics of list comprehensions in Erlang, tidier has to restrict itself to funs whose argument is a variable. For example, without precise information about the types of the list elements it is *not* permitted to perform the following *list map to comprehension* refactoring:

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

⇓

```
[X + Y || {X, Y} <- L]
```

because the former code will raise an exception if the list contains some element other than a pair, while the latter will simply filter out this element. Similar constraints hold also for transforming `lists:filter/2`, even though, as we will see below, we can often do better in this case.

Inlining simple boolean filtering funs

The most commonly occurring fun used in a `lists:filter/2` is a fun consisting of two clauses. The first clause, which is usually the `true` branch, has a specific clause head pattern and possibly also a set of guards (either in the clause head or in the body) specifying which list elements to keep. The second is a match-all clause to filter out all other elements.

For such list filtering funs, tidier's refactoring uses the head pattern as a filter expression in the list comprehension generator, and the guards (if any) as further filters after the generator. We illustrate this refactoring with a code fragment from Erlang/OTP R13B's `lib/appmon/src/appmon_dg.erl:69`:

```
efilter(Es) ->
    lists:filter(fun ({_V1, _V2, primary}) -> true;
                     (_E) -> false
                 end, Es).
```

⇓

```
efilter(Es) ->
    [E || E = {_V1, _V2, primary} <- Es].
```

and with a clause from `lib/asn1/src/asn1ct.erl:2436` (but with the actual function name abbreviated):

```
gff(_, Name, L) when is_atom(Name); is_list(Name) ->
    lists:filter(fun ({N,_,_}) when N == Name -> true;
                     (_) -> false
                 end, L);
```

⇓

```
gff(_, Name, L) when is_atom(Name); is_list(Name) ->
    [T || T = {N, _, _} <- L, N == Name];
```

In both cases, we have taken the liberty to also use the *static structure reuse* refactoring we are going to present in Section 4.6.

4.5 Transformations requiring type information

Some refactorings require or benefit from type information. We describe those that tidier currently implements.

Specializing the size function

Till quite recently, there was only one way to find the size of a tuple or a binary: by employing the overloaded function `size/1`, which could also be used as a guard. Consequently, many programs have been written using this function. Erlang/OTP R12 introduced two specializations of this function: `tuple_size/1` which works with tuples only and `byte_size/1` which works with bitstrings (binaries are just a special case of bitstrings). These functions are preferable because they express in a better way the intention of the programmer, provide more information to static analysis tools such as Dialyzer [9], and are slightly more efficient than `size/1`. Unfortunately, manual conversion of existing programs is both tedious and error prone. Tidier comes to the rescue here: it employs local type inference to determine the type of `size`'s argument and specializes the call appropriately. At least in the code of Erlang/OTP, we have seen only few cases where the inference is not strong enough to automatically perform this specialization. These cases are left for manual refactoring.

Simplifying guard sequences

This refactoring started because we noticed that, especially with `size/1` being overloaded, it was quite common for tidier to come across code that looks as follows:

```
foo(T) when is_tuple(T), size(T) > 2 -> ...
```

The *size specialization* refactoring of the previous paragraph will transform this code to:

```
foo(T) when is_tuple(T), tuple_size(T) > 2 -> ...
```

and it is pretty easy to notice now that the `is_tuple/1` guard is semantically not needed anymore, because the `tuple_size/1` guard does not succeed for anything but tuples. Consequently the code can be simplified to the following:

```
foo(T) when tuple_size(T) > 2 -> ...
```

Once this refactoring was in place, we decided to extend it to simplify other guard sequences that Erlang programmers occasionally write most probably unaware that they are unnecessarily cluttering their code with tests which are implied by others.

```
foo(Rec, Fields, Key) when is_tuple(Rec), is_list(Fields),
                           size(Rec)-1 =:= length(Fields) ->
    lists:zip([Key|Fields], tuple_to_list(Rec)).
```
$\Longrightarrow$
```
foo(Rec, Fields, Key)
  when tuple_size(Rec)-1 =:= length(Fields) ->
    lists:zip([Key|Fields], tuple_to_list(Rec)).
```

Figure 3. A guard simplification refactoring from actual code (`apache-couchdb-0.8.1/src/mochiweb/mochiweb_util.erl:422`).

```
decode_octets(<<0:1,Len:7,Bin/binary>>, C, Acc) ->
  <<Value:Len/binary-unit:8,Bin2/binary>> = Bin,
  BinOctets = list_to_binary(reverse([Value|Acc])),
  case C of
    Int when is_integer(Int), size(BinOctets) == Int ->
      {BinOctets,Bin2};
    ...
```
$\Longrightarrow$
```
decode_octets(<<0:1,Len:7,Bin/binary>>, C, Acc) ->
  <<Value:Len/binary-unit:8,Bin2/binary>> = Bin,
  BinOctets = list_to_binary(reverse([Value|Acc])),
  case C of
    Int when byte_size(BinOctets) =:= Int ->
      {BinOctets,Bin2};
    ...
```

Figure 4. Another guard simplification refactoring from actual code (`lib/asn1/src/asn1rt_per_bin.erl:495`).

Two examples Figure 3 shows one interesting such case from the code of CouchDB. The first two guards are unnecessary as they are implied by the third once the `size/1` guard has been specialized.

Similarly, Figure 4 shows a case from the code of the `asn1` application of Erlang/OTP R13B. Given built-in knowledge that the return type of `size` functions is integer, the guard sequence can be simplified.[2]

4.6 Transformations that eliminate redundancy

As the astute reader has no doubt noticed from the examples of the previous section, there is a fine line between code simplification refactorings and transformations that an optimizing compiler performs. Tidier further explores this idea and offers some refactorings that are partly inspired by compiler optimizations.

Avoid re-creation of existing tuples and lists

In Erlang identical tuples or lists created in different points of a clause, where one point dominates the other, can be assigned to variables and subsequently become shared, thereby avoiding their unnecessary re-creation. This refactoring, called *static structure reuse*, is illustrated below:

```
t({X, [3, Y]}) ->
  case m:foo(X) of
    true ->
      [3, Y];
    false ->
      {X, [3, Y]}
  end.
```
$\Longrightarrow$
```
t({X, [3, _Y] = L} = T) ->
  case m:foo(X) of
    true ->
      L;
    false ->
      T
  end.
```

This is exactly what tidier would do in this case. The notion of identity that tidier uses to identify opportunities for this refactoring is *syntactic identity*: i.e., two structures are considered identical if they have exactly the same statically known sub-terms, including the same variable names, in all their corresponding positions. Note however that these sub-terms cannot contain function calls because these calls may invoke side-effects.

The main advantages of this refactoring are that it typically makes the source code shorter and its execution more efficient both in time and in space. Indeed, many Erlang programmers who are aware of its benefits perform this refactoring by hand on their programs.[3] However, it is often quite difficult for the human eye to spot all opportunities for structure reuse in programs, especially those that are not immediately obvious. For example we have noticed that, even in code of performance conscious programmers, the following case of deconstructing and constructing the same term typically remains untransformed:

```
[{A, B, C, D} || {A, B, C, D} <- List]
```

The *static structure reuse* refactoring of tidier transforms the above to:

```
[T || T = {_A, _B, _C, _D} <- List]
```

which is both shorter and will execute more efficiently, both in time and in space. (The BEAM bytecode compiler currently does not perform this optimization and will create copies of the tuples for the list comprehension's result.)

On the other hand, a problem with this refactoring is that if performed aggressively, as an optimizing compiler performing *common subexpression elimination* would do it, it results in code which is quite unnatural and, in all probability, would not be something performed also by a human programmer. This is especially true for lists and we illustrate it by the following example:

```
t([X, Y, Z]) ->
  case m:foo(X) of
    true ->
      [Z];
    false ->
      [Y, Z]
  end.
```
$\not\Longrightarrow$
```
t([X | [Y | [Z] = L1] = L2]) ->
  case m:foo(X) of
    true ->
      L1;
    false ->
      L2
  end.
```

Since only few programmers would consider the code on the right an improvement over the one on the left as far as code readability is concerned, tidier does *not* perform such refactorings. In particular, the static structure reuse refactoring treats lists as atomic objects and never breaks them into smaller parts.

Simplifying control

Refactorings under this category involve `case`s and `if`s and come in two flavours: *straightening statements* and *simplifying (matching or logical) expressions*.

Straightening Sometimes, perhaps due to code evolution, control statements can end up having only one alternative and this refactoring straightens their code. This is illustrated in Figure 5. It is clear that the code becomes smaller and actually in this case it is also more uniform in style. The only side-effect, albeit a relatively innocent one, is that this code might raise a `badmatch` rather than a `case clause` exception if `Reply` is not an `ok`-tagged pair.

[2] Actually, a global type analysis would discover that the `is_integer/1` guard is completely redundant in the code of Figure 4.

[3] The inclusion on the list of refactorings performed by tidier of the *structure reuse* refactoring was a suggestion to us by Kenneth Lundin.

```
...
case Reply of
  {ok, Socket} ->
     {ok, {IP, _Port}} = inet:peername(Socket),
     true = member_address(IP, which_slaves()),
     PS = erl_prim_loader:prim_init(),
     boot_loop(Socket, PS)
end.
```

$\Longrightarrow$

```
...
{ok, Socket} = Reply,
{ok, {IP, _Port}} = inet:peername(Socket),
true = member_address(IP, which_slaves()),
PS = erl_prim_loader:prim_init(),
boot_loop(Socket, PS).
```

Figure 5. An example of `case` straightening (from R13B's `lib/kernel/src/erl_boot_server.erl:274`).

Sometimes, the source code has clear signs that the control flow of the `case` statement is intentional as in the code shown below:

```
case mod:has_property(X) of
  true -> handle(X)
  %% all other cases not handled yet
  %% false -> ...
  %% unknown -> ...
end,
```

Since tidier cannot read comments (or the minds of programmers!), as a rather ad hoc heuristic, it will never perform straightening on code that has a comment inside a `case` statement.

Simplifying expressions The `case` expression in Erlang is a powerful construct, but occasionally some `case` expressions clutter the code unnecessarily. The following is an example from the source code of Erlang/OTP R13B's `lib/kernel/src/group.erl:368`.

```
case get_value(binary, Opts, case get(read_mode) of
                               binary -> true;
                               _ -> false
                             end) of
   true -> ...
```

Tidier simplifies the above code to:

```
case get_value(binary, Opts, get(read_mode) =:= binary) of
   true -> ...
```

As another, rather interesting example of unnecessary code cluttering, we show the refactoring of code from Erlang/OTP R13B's `lib/xmerl/src/xmerl_ucs.erl:549`. (The function name and one variable name are shorter so that the example fits here.)

```
t_charset(Fun, In) ->
  case lists:all(Fun, In) of
    true ->
      true;
    _ ->
      false
  end.
```

$\Rightarrow$

```
t_charset(Fun, In) ->
  lists:all(Fun, In).
```

Such refactorings are aided by tidier having knowledge about the return type of commonly employed functions; e.g., that the return value of `lists:all/2` is either `true` or `false`. Similar cases, involving the `lists:member/2` function, occur in the code of `lib/inviso/src/inviso_tool_lib.erl:342` and in the code of `lib/inviso/src/inviso_tool.erl:2125`.

Switching on `true` and `false` is very common and this programming idiom often clutters the code unnecessarily. The clause on Figure 6 is from `lib/hipe/cerl/cerl_to_icode.erl:2370` and is simplified as shown in the figure.

Naturally, such simplifications are not restricted to `case` expressions, but are also applicable to `if`s. The following example is from the code of `lib/percept/src/egd_render.erl:313`.

```
if
   Yp =:= Y -> true;
   true -> false
end
```

$\Longrightarrow$

```
Yp =:= Y
```

```
is_pure_op(N, A) ->
    case is_bool_op(N, A) of
        true -> true;
        false ->
           case is_comp_op(N, A) of
               true -> true;
               false -> is_type_test(N, A)
           end
    end.
```

$\Downarrow$

```
is_pure_op(N, A) ->
    is_bool_op(N, A) orelse is_comp_op(N, A)
                     orelse is_type_test(N, A).
```

Figure 6. Simplification of nested `case` expressions.

Many other similar expression simplifications are currently automatically performed by tidier. We will see in the next section how such simplifications come in handy in creating better looking list comprehensions.

4.7 Simplifying list comprehensions even further

Having the ability to simplify expressions allows us to do more effective transformations of maps and filters to list comprehensions. For example, consider the following code:

```
lf(X, List) ->
  lists:filter(fun (Y) ->
                 if
                    X =:= Y -> true;
                    true -> false
                 end
               end,
               List).
```

By combining the power of the refactorings we have shown, the code can be simplified to:

```
lf(X, List) ->
  [Y || Y <- List, X =:= Y].
```

While the above example is fictitious, it does not differ much from actual Erlang code that tidier has identified as simplifiable. For example, the code of `lib/kernel/src/pg2.erl:280` in Erlang/OTP R13B reads:

```
lists:filter(fun(Pid) when node(Pid) =:= Node -> false;
                (_) -> true
             end,
             Pids)
```

Tidier automatically transforms the above code to:

```
[Pid || Pid <- Pids, node(Pid) =/= Node]
```

Similarly, the code of `src/web/ejabberd_http_bind.erl:956` from Ejabberd 2.0.1 reads:

```
lists:filter(fun (I) ->
               case I of
                 {xmlelement, _, _, _} -> true;
                 _ -> false
               end
             end,
             Els)
```

and is automatically transformed by tidier to:

```
[I || I = {xmlelement, _, _, _} <- Els]
```

Once this functionality was in place, it whetted our appetite for more. Unfortunately, to do considerably more requires information from a global type analysis. Writing such an analysis and hooking tidier to it is currently future work. However, we noticed that in some cases even a simple function-local type analysis can provide sufficient information for what we wanted to do. This is especially true when calls to `lists:map/2` and `lists:filter/2` are nested within each other. Currently tidier handles this case specially, effectively performing *deforestation* [17] at the level of source code. Some of the cases we found in real code are interesting and worth the effort. Let's see some examples.

The code of `lib/inviso/src/inviso_tool_sh.erl:1638` from Erlang/OTP R13B (when pretty-printed) reads:

```
get_all_tracing_nodes_rtstates(RTStates) ->
  lists:map(fun ({N,_,_}) -> N end,
            lists:filter(fun ({_,{tracing,_},_}) -> true;
                             (_) -> false
                         end,
                         RTStates)).
```

Tidier automatically transforms this code to:

```
get_all_tracing_nodes_rtstates(RTStates) ->
  [N || {N,{tracing,_},_} <- RTStates].
```

The transformation is correct since the `lists:filter/2` call provides sufficient type information, namely that the intermediate list will consist of triples only, which guarantees that the `lists:map/2` call will not throw an exception.

A case similar to the above also occurs in the code of Wrangler (`src/refac_rename_fun.erl:344`):

```
lists:map(fun ({_, X}) -> X end,
          lists:filter(fun (X) ->
                         case X of
                           {atom, _X} -> true;
                           _ -> false
                         end
                       end,
                       R))
```

Tidier transforms this code to:

```
[X || {atom, X} <- R]
```

In both cases, the code is not only considerably more readable but also more efficient as the input list is traversed only once and no intermediate list is constructed.

4.8 List comprehensions in conjunction with zip and unzip

One case that is currently treated specially by tidier is when the list that will become the generator of a list comprehension is a list produced by a call to `lists:zip/2`, which produces a list of pairs from two lists. The following example is also from the code of Wrangler (`src/refac_annotate_pid.erl:274`):

```
lists:map(fun ({A, P}) -> F(A, P) end,
          lists:zip(Args, ParSig))
```

Having built-in type information about the result of `lists:zip/2` being a list of pairs, allows tidier to currently transform the above code to the following:

```
[F(A, P) || {A, P} <- lists:zip(Args, ParSig)]
```

However, our plan is that if the *comprehension multigenerators* Erlang Enhancement Proposal (EEP-19 [14]) is accepted and implemented in Erlang/OTP, tidier will transform the above case to:

```
[F(A, P) || A <- Args && P <- ParSig]
```

thereby avoiding the construction of the intermediate list.

Since the case of `lists:zip/2` was treated specially, it felt natural that tidier should also pay some attention to `lists:unzip/1`. The following is an interesting example of a significant simplification of actual code that tidier currently performs (from Nokia's `tuulos-disco-0.1/master/src/event_server.erl:123`):

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

Tidier simplifies the above code to:

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

thereby completely eliminating the construction of the list of pairs, and its deconstruction by the `lists:unzip/1` call.

We have seen enough examples of transformations performed by tidier. We stress again that all these refactorings are performed in a completely automatic way by tidier. Let us now briefly see how tidier can be used.

5. Tidier at Work

For those not faint at heart, 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 source code of `myfile.erl` and overwrite the contents of the file with the resulting source code. Multiple source files can also be given. Alternatively, the user can tidy a whole set of applications by a command of the form:

```
> tidier dir1 ... dirN
```

which will tidy all `*.erl` files 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 --comprehensions --size myfile.erl
```

to only transform uses of `lists:map/2` and `lists:filter/2` to list comprehensions and uses of `size/1` to `tuple_size/1` or `byte_size/1`. We refer the reader to tidier's manual for the complete set of command-line options.

A very handy option is the option that will cause tidier to just print on the standard output the list of transformations that

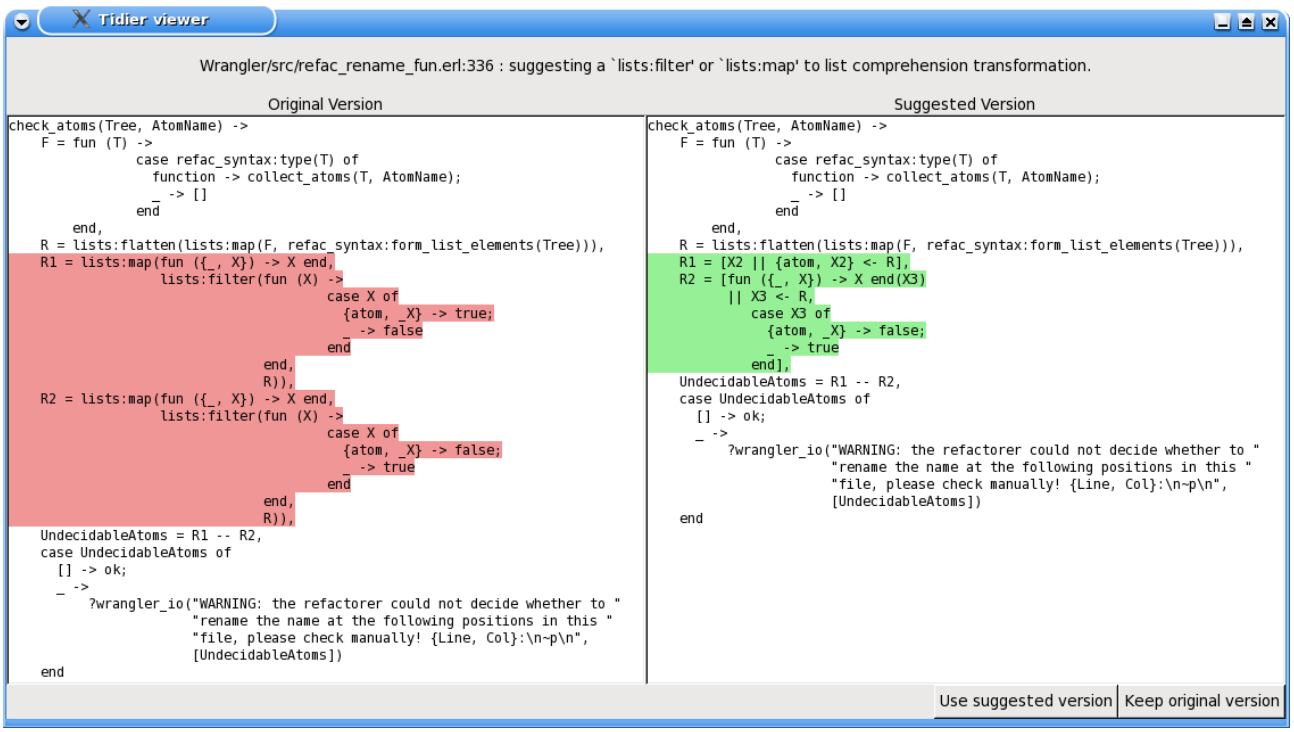

Figure 7. Tidier in action: simplifying the code of Wrangler.

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 tidier users will probably prefer this mode of using tidier, at least initially.

Figure 7 shows tidier's GUI in action. In fact, the snapshot depicts tidier refactoring the `src/refac_rename_fun.erl` file from the code of the current development branch of Wrangler. Tidier has identified two opportunities for transforming maps and filters to list comprehensions. The first combination of map and filter is the example we discussed in Section 4.7. As can be seen in the snapshot, that code has been changed to a one-liner. The second map and filter combination cannot be simplified to the same extent and still preserve its semantics. Tidier's GUI shows the old code (on the left) and the new code (right); the code parts that differ between the two versions are coloured appropriately. At this point, the user can either press the "Use suggested version" button to accept tidier's refactoring 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).

Current experiences

As it is probably obvious by now, during its development, tidier has been repeatedly applied to large code bases; most notably to the source code of Erlang/OTP, currently consisting of about 1,200,000 lines of Erlang code. As a side comment, on a relatively recent desktop, tidier is able to virtually refactor all this code (i.e., just detect and print out the list of transformations that would be performed on these files) in about two and a half minutes. On those Erlang/OTP libraries that we are directly involved in their development or have permission to change them, tidier's suggestions have been adopted. Moreover, on those libraries that our group is responsible for, tidier is now part of the tools used for their development and is run periodically over their code.

We have also applied tidier on various open source and often widely used applications written in Erlang (Apache CouchDB, ejabberd, Erlang Web, RefactorErl, Scalaris, Wings, Wrangler, Yaws, etc.), totalling about 300,000 lines of Erlang code. A detailed experience report on using tidier on them is beyond the scope of the current paper but instead is described in a companion paper [2]. It suffices to say that there are plenty of opportunities for modernizing Erlang code out there, eliminating various bad code smells, automatically cleaning up source code of applications and simplifying it. Overall, given the ease of use of tidier, we see few reasons not to try it out and adopt most of its suggestions.

6. Related Work

Software refactoring [4], the process of restructuring an existing body of program code in order to alter its internal structure and improve its readability and maintainability without changing its external behaviour, is by now an established and well-researched technique in many programming languages. Especially in object-oriented languages, refactoring is supported by a number of tools such as editors, IDEs, and *refactoring browsers*; see the survey by Mens and Tourwé [12] and the references therein.

In the context of declarative languages, although program transformation is a well-researched area by now, explicit tool support for refactoring programs at the level of source code is less common. Besides tidier, notable exceptions of semi-automatic code refactoring tools are the HaRe tool for Haskell [8], the ViPReSS tool for Prolog [16], and the RefactorErl and Wrangler tools for Erlang. The last two tools we review in more detail below.

RefactorErl is an Erlang refactoring tool, developed by researchers at the Eötvös Loránd University in Budapest, Hungary, that aims to assist Erlang programmers perform semi-automatic refactoring of their code. The tool follows a disciplined approach to refactoring and works by creating a formal semantical graph model from

Erlang source code and storing this graph in a relational database. This graph can be modified on the syntax tree level and the source code is reproducible from there. The RefactorErl tool comes with a user interface provided as an Emacs minor mode to help programmers perform a predefined set of refactoring transformations. Some of these refactorings are very simple (e.g., rename a variable or a record). Some other refactorings are more sophisticated and can for example be used to change uses of tuples to records in some module [11] or refactor the module structure of an existing application by using code clustering techniques [10]. However, it is unclear to what extent the more sophisticated refactorings are available in the public release of RefactorErl at the time of this writing.

Wrangler is a more mature Erlang refactoring tool, developed by Huiqing Li and Simon Thompson at the University of Kent, U.K. The tool supports the interactive refactoring of Erlang programs under both Emacs and ErlIDE (the Erlang plug-in for Eclipse), and is publicly available under an open source licence. Wrangler supports various semi-automatic data and process refactorings [7] and quite recently has also been enhanced with the ability to detect and remove duplicated code [6]. All these refactorings are initiated and controlled by the programmer. According to a published survey of Erlang tools [13] conducted in the spring of 2008, Wrangler was moderately well-known in the Erlang community (33%) but not used much (5%), although the situation may of course have changed by now.

Compared with these refactoring tools for Erlang, tidier differs significantly both in the kind of refactorings that it performs but, more importantly, also in design philosophy. In its primary mode of operation, tidier is fully automatic and requires no interaction from its user. As such, tidier needs to provide strong guarantees of preservation of semantic equivalence between the original and transformed program and cannot afford to leave this responsibility on the programmer. On the other hand, this means that tidier's refactorings are more limited in scope (currently, they are mostly clause-local) than those of RefactorErl or Wrangler which can perform module-scope or even application-wide refactorings. Still, we think that some of tidier's refactorings are very interesting.

Perhaps surprisingly, with the exception of the ReSharper [15] add-in to Visual Studio, we were not able to locate any other fully automatic code cleanup tools in any high-level language.[4] We hope that tidier will pave the way for more fully automatic code simplification and cleanup tools in Erlang and other languages.

7. Concluding Remarks

In this paper we described tidier, a software tool that automatically tidies Erlang source code, making it cleaner, simpler, and often also more efficient. In doing so, tidier not only simplifies and modernizes the code of an application, but also increases its readability, maintainability and often performance. We strongly believe that the ease of use of our tool makes tidier attractive to use in any Erlang project, if not as an automatic refactorer, at least as a detector of bad code smells in the code. Alternatively, tidier's GUI can be used in existing Erlang code bases to illustrate to programmers excerpts of existing code that could be written more elegantly or simply.

In this respect, our paper is interesting to its community not only as a tool description paper but also as a catalog of good coding practices, some of which are publicly documented for the first time.

We stress that the paper described the architecture and current status of our tool. Various additions to tidier's functionality are already planned; their priority might change based on feedback that we may receive from users of our tool after its first public release.

[4] There are of course plenty of tools that automatically indent source code of many languages or automatically cleanup and/or validate HTML pages.

Acknowledgements

Although by now there are relatively few remains of `erl_tidy`'s original code in the source code of tidier, the `erl_tidy` module of the `syntax_tools` library has served both as inspiration and as a very good starting point for the development of tidier. We thank its author, Richard Carlsson, both for releasing his code and for the comments and suggestions that he sent us.

References

[1] J. Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN Conference on History of Programming Languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[2] T. Avgerinos and K. Sagonas. Cleaning up Erlang code is a dirty job but somebody's gotta do it. In *Proceedings of the Eighth ACM SIGPLAN Erlang Workshop*, New York, NY, USA, Sept. 2009. ACM.

[3] R. Carlsson. Syntax tools reference manual, version 1.6, Apr. 2009. `http://www.erlang.org/doc/apps/syntax_tools/`.

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

[5] P. Gustafsson and K. Sagonas. Bit-level binaries and generalized comprehensions in Erlang. In *Proceedings of the Fourth ACM SIGPLAN Erlang Workshop*, pages 1–8, New York, NY, USA, Sept. 2005. ACM.

[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] H. Li, S. Thompson, and C. Reinke. Tool support for refactoring functional programs. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 27–38, New York, NY, USA, Aug. 2003. ACM.

[9] 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.

[10] 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.

[11] L. Lövei, Z. Horváth, T. Kozsik, and R. Király. Introducing records by refactoring. In *Proceedings of the 6th ACM SIGPLAN Workshop Erlang*, pages 18–28, New York, NY, USA, Sept. 2007. ACM.

[12] T. Mens and T. Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, Feb. 2004.

[13] T. Nagy and A. Nagyné-Víg. Erlang testing and tools survey. In *Proceedings of the 7th ACM SIGPLAN Workshop on Erlang*, pages 21–28, New York, NY, USA, Sept. 2008. ACM.

[14] R. A. O'Keefe. Erlang Enhancement Proposal: Comprehension multigenerators, Aug. 2008. `http://www.erlang.org/eeps/eep-0019.html`.

[15] ReSharper 4.5. `http://www.jetbrains.com/resharper/`.

[16] A. Serebrenik, T. Schrijvers, and B. Demoen. Improving Prolog programs: Refactoring for Prolog. *Theory and Practice of Logic Programming*, 8(2):201–215, Mar. 2008.

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