Εθνικο Μετσοβιο Πολυτεχνειο
Τμημα Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Λογισμικού

Αυτόματη Απλοποίηση και Αναδιαμόρφωση Προγραμμάτων σε Erlang

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αθανάσιου Ι. Αυγερινού

Επιβλέπων: Κωνσταντίνος Φ. Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗ-
ΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Λογισμικού

Αυτόματη Απλοποίηση και Αναδιαμόρφωση Προγραμμάτων σε Erlang

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Αθανάσιου Ι. Αυγερινού

Επιβλέπων: Κωνσταντίνος Φ. Σαγώνας
Αναπληρωτής Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 14^{η} Ιουλίου 2009.

....	....	....
Κωνσταντίνος Σαγώνας	Νικόλαος Παπασπύρου	Ευστάθιος Ζάχος
Αναπληρωτής Καθηγητής Ε.Μ.Π.	Επίκουρος Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2009

....
Αθανάσιος Ι. Αυγερινός
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Στην παρούσα διπλωματική περιγράφουμε τους σχεδιαστικούς στόχους και την τρέχουσα κατάσταση του tidier, ενός εργαλείου λογισμικού που αναδιαμορφώνει πηγαίο κώδικα Erlang, κάνοντάς τον καθαρότερο, απλούστερο και σε πολλές περιπτώσεις και πιο αποδοτικό. Σε αντίθεση με άλλα εργαλεία αναδιαμόρφωσης, ο tidier είναι πλήρως αυτόματος και ανεξάρτητος από άλλες εφαρμογές γραφής κώδικα. Το εν λόγω εργαλείο παρέχει μια ευρεία γκάμα μετασχηματισμών, οι οποίοι μπορούν να επιλεχθούν μέσω ορισμάτων από την γραμμή εντολών και να εφαρμοστούν σε ένα σύνολο αρχείων ή και ολόκληρες εφαρμογές με μία απλή εντολή. Εναλλακτικά, οι χρήστες μπορούν να χρησιμοποιήσουν το γραφικό περιβάλλον που παρέχει ο tidier ώστε να επιβλέπουν ένα προς ένα τους μετασχηματισμούς που πραγματοποιούνται στον κώδικά τους και να επιλέγουν μόνο αυτούς που επιθυμούν. Ο tidier έχει ήδη χρησιμοποιηθεί για να αναδιαμορφωθούν διάφορες εφαρμογές του Erlang/OTP και έχει δοκιμαστεί σε πολλές σημαντικού μεγέθους εφαρμογές Erlang ανοικτού κώδικα. Αναφέρουμε τις εμπειρίες μας και παρουσιάζουμε ευκαιρίες για να εφαρμοστούν οι τρέχοντες μετασχηματισμοί του tidier σε υπάρχοντα κώδικα Erlang. Ως επακόλουθο, σε αυτήν την διπλωματική περιγράφονται και ποιες πρακτικές οδηγούν σε ποιοτικό κώδικα Erlang. Τέλος, περιγράφουμε λεπτομερώς την αυτοματοποιημένη μεθοδολογία αναδιαμόρφωσης κώδικα που υποστηρίζουμε και ένα σύνολο μετασχηματισμών που είναι αρκετά γενικοί ώστε να μπορούν να εφαρμοστούν ως έχουν ή με μικρές παραλλαγές σε προγράμματα γραμμένα σε Haskell ή Clean και ίσως ακόμα και σε μη συναρτησιακές γλώσσες προγραμματισμού.

Λέξεις Κλειδιά

μετασχηματισμός προγράμματος, αναδιαμόρφωση κώδικα, εκκαθάριση κώδικα, απλοποίηση κώδικα, Erlang

Abstract

This thesis 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 report our experiences and show opportunities for tidier's current set of transformations on existing Erlang code out there. As a by-product, this thesis also documents what we believe are good coding practices in Erlang. Last but not least, we describe 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.

Keywords

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

Ευχαριστίες

Καταρχάς θα ήθελα να ευχαριστήσω τον άνθρωπο χάρη στον οποίο υπάρχει αυτή η διπλωματική και έχει αυτήν τη μορφή. Με το συνεχές ενδιαφέρον του, τον αστείρευτο ενθουσιασμό του και την πάντα ανοιχτή του πόρτα προς τους φοιτητές αποτελεί μια έμπνευση για μένα. Ελπίζω κάποια στιγμή να μπορώ να προσφέρω και εγώ όσα προσφέρεις στους φοιτητές σου. Η συνεργασία μας τον τελευταίο χρόνο ήταν μια υπερπολύτιμη εμπειρία για μένα και ξέρω ότι θα μου λείψει πολύ. Ένα τεράστιο ευχαριστώ στον καθηγητή, καθοδηγητή και φίλο μου Κωστή Σαγώνα.

Θα ήθελα επίσης να ευχαριστήσω τους καθηγητές Νίκο Παπασπύρου και Στάθη Ζάχο. Ήταν οι άνθρωποι που με οδήγησαν στα πρώτα βήματά μου στον χώρο της πληροφορικής και αποτέλεσαν τον λόγο που αποφάσισα να ακολουθήσω αυτήν την κατεύθυνση. Καθένας από εσάς αποτελεί ένα πρότυπο για μένα, όχι μόνο σε επίπεδο καθηγητή αλλά και ανθρώπου. Θέσατε τον πήχυ πολύ ψηλά. Σας ευχαριστώ για όλα.

Ένα θερμό ευχαριστώ χρωστάω επίσης σε όλους τους συμφοιτητές και φίλους μου με τους οποίους πέρασα όλα αυτά τα χρόνια. Σας ευχαριστώ για όλες αυτές τις υπέροχες στιγμές που μου χαρίσατε, κάνοντας τα 5 χρόνια του πολυτεχνείου την πιο υπέροχη εμπειρία της ζωής μου. Εύχομαι και ελπίζω να μπορέσουμε να διατηρήσουμε τις επαφές μας και μετά το πολυτεχνείο.

Τέλος και πάνω από όλα, θέλω να ευχαριστήσω την οικογένειά μου. Χωρίς εσάς τίποτα από όλα αυτά δεν θα ήταν εφικτό. Χάρη στην διαρκή σας στήριξη, αγάπη, βοήθεια και ενθάρρυνση βρίσκω το κουράγιο να συνεχίσω. Είστε αυτό που θα μου λείψει περισσότερο από την φυγή μου προς το εξωτερικό. Δεν υπάρχουν λόγια για να εκφραστώ. Το μόνο που μπορώ και κάνω είναι να σας προσφέρω την αγάπη μου. Ευχαριστώ.

Θανάσης Αυγερινός

Contents

List of Tables

List of Figures

Preface

This diploma thesis is the compilation and adaptation of two papers [2, 22] which are about to appear in international conference proceedings. In compiling them we have tried to remove the duplicated material, enrich various sections with more detailed information and include some of our most recent work. Some familiarity with functional programming and preferably Erlang is expected for chapter 4.

Chapter 1

Introduction

1.1 Motivation

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. 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 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, my advisor, both due to an obsession with code cleanliness and 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 also 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.

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.

1.2 Outline of the thesis

To make the thesis relatively self-contained, the next chapter briefly discusses the idea of refactoring and overviews the Erlang language and the evolution of its implementation. Chapter 3 presents the design characteristics and main properties of tidier. It is followed by the main chapter of this thesis, Chapter 4, that describes in detail the code transformations currently performed by the tool. In Chapter 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. Chapter 6 reviews related work and the thesis ends with some concluding remarks. Appendix A contains a reference manual of the refactoring tool.

Chapter 2

Preliminaries

2.1 Refactoring

According to [8], refactoring is the process of changing a computer program's internal structure without modifying its external functional behavior or existing functionality. The main purpose of refactoring is to make the software easier to understand and modify. Although refactoring – as we know it today – is a relatively new notion, it is becoming increasingly important in several fields and especially software engineering. In extreme programming and other agile methodologies, refactoring is an integral part of the software development cycle: developers first write tests, then write code to make the tests pass, and finally refactor the code to improve its internal consistency and clarity. Automatic unit testing helps to preserve correctness of the refactored code. The refactoring research area is still relatively young and unexplored and most of the related work 6.1 is very recent.

Why is refactoring important Refactorings can be used to improve several important internal quality attributes of software, such as:

- *software design and structure.* In every software project of significant size it very difficult or even impossible to finalize the design and structure from the beginning. During the development and maintenance of the project, most probably the structure will have to change quite a lot of times. However, such unorganized changes may result in software structure of poor quality. Frequent refactoring can help improving software design in an efficient and controlled manner.

- *code readability.* Having code that is cluttered with redundancies and obfuscated expressions is something not uncommon among most programmers. This is usually a result of software evolution (new features are added, bugs are fixed etc) and can be also addressed with cautious refactoring.

- *maintainability.* Performing refactorings to improve code readability and design has as an immediate effect an improvement over code maintainability. Having well-organized and simple, uncluttered code can help the developer have a better understanding of the code and therefore perform maintenance tasks (like bug-finding and bug-fixing) much faster.

- *adhering to a specific programming paradigm.* High level programming languages may provide a lot of ways to express the programmer's intention. Code refactoring can be employed to transform code segments from one paradigm to another (when one of them is encouraged). Some of the refactorings that we will explore in this thesis belong to a variation of this category.

- *extensibility.* Software extensibility can be significantly improved as a side-effect of performing refactoring to improve the design and structure of software (adding new features becomes much easier and faster).

- *performance.* Although refactorings can usually lead to a decrease in performance (in favour of code design and readability), better-structured code can be optimized much more effectively than poorly-designed software during a performance optimization stage. Moreover, not all refactorings result in a decrease in performance. Specifically, none of the refactorings that are described in this thesis (and are applied by `tidier`) decreases performance. Instead, many of the suggested refactorings can lead to better performance.

Let us now see the language which we will employ to apply our refactorings.

2.2 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 2.1 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].

```
-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 2.1: An example Erlang program.

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 to perform pattern matching directly on *binaries* and *bit streams* [9] 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 run-time 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 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 in the following chapter.

Chapter 3

Design

3.1 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 [12] or RefactorErl [15].

3.1.1 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.

3.1.2 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.

[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.

Chapter 4

Transformations

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. Moreover, most of the transformations that are not trivial contain a small paragraph that briefly addresses some of the semantics-preserving details of the refactoring.

4.1 Simple transformations

We start by describing the simplest transformations. Some of these transformations (and some of Section 4.3) are also provided by the `erl_tidy` module which we used as a starting point for tidier. The refactorings described in this section are quite simple and do not affect the operational semantics of the program being transformed.

4.1.1 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.

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 and replace or with 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.

4.1.2 Eliminating explicit imports

This transformation eliminates all import statements and rewrites all calls to explicitly imported functions as remote calls. It is illustrated below:

```
-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 becomes 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.

4.1.3 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.

4.1.4 Turning funs to functions

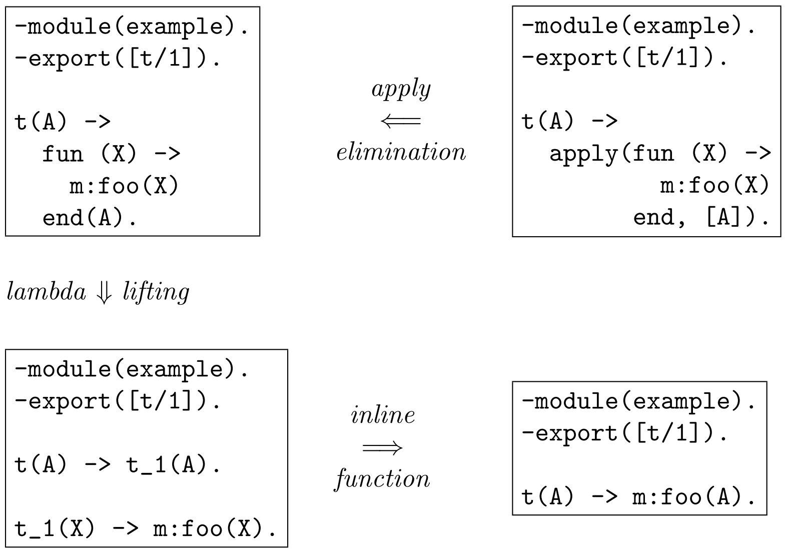
Lambda lifting or closure conversion [10] is the process of eliminating free variables from local function definitions from a computer program. The elimination of free variables allows the compiler to hoist local definitions out of their surrounding contexts into a fixed set of top-level functions with an extra parameter replacing each local variable. By eliminating the need for run-time access-links, this may reduce the run-time cost of handling implicit scope. Many functional programming language implementations use lambda lifting during compilation. This compiler optimization served as an inspiration for this refactoring which transforms fun expressions to local functions and changes the point where the fun expression was applied to a function call. For the reader that is familiar with object-oriented programming languages this transformation is actually quite similar to the well-known *extract method* refactoring [8].

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 4.1 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. As we will see in the following sections, many of the most advanced refactorings benefit or are triggered by the simple refactorings of this section.

4.1.5 Introducing `min` and `max`

In a programming language that is still under development and constantly evolves it is very common for newly introduced library functions to allow a simpler



```
-module(example).
-export([t/1]).

t(A) ->
  fun (X) ->
    m:foo(X)
  end(A).
```

```
-module(example).
-export([t/1]).

t(A) ->
  apply(fun (X) ->
            m:foo(X)
        end, [A]).
```

```
-module(example).
-export([t/1]).

t(A) -> t_1(A).

t_1(X) -> m:foo(X).
```

```
-module(example).
-export([t/1]).

t(A) -> m:foo(A).
```

Figure 4.1: Illustration of refactorings that work hand in hand.

and more succinct way of expressing the programmer's intention. Erlang could be no exception and in this section we will present an automatic refactoring that employs the newly introduced library functions `erlang:min/2` and `erlang:max/2` in order to simplify existing code.

Although Erlang has always supported a total order among terms there was no `min` or `max` function (the aforementioned library functions have only just been introduced). As a result, there is a huge number of erlang modules that contain a local `min` or `max` function. Moreover, the absence of these functions has cluttered many pieces of code with expressions of the form (this is actual code from Erlang/OTP R13B's `otp/hipe/util/hipe_dot.erl:145`):

```
calc_dim([$\\, $n|T], H, TmpW, MaxW) ->
   if TmpW > MaxW -> calc_dim(T, H+1, 0, TmpW);
      true -> calc_dim(T, H+1, 0, MaxW)
   end
```

By employing the *min-max* refactoring, tidier will immediately transform the above to:

```
calc_dim([$\\, $n|T], H, TmpW, MaxW) ->
   calc_dim(T, H+1, 0, erlang:max(TmpW, MaxW))
```

Although this is a rather trivial refactoring, we present it here to show how tidier cannot only be used to modify existing code, but also to easily introduce new functions that can make the code simpler and of better quality.

4.1.6 Code beautification

Tidier, due to its automatic nature can also be used to save the programmer's precious time by performing all sorts of tedious tasks. As an example we will present a transformation that was suggested to us by a programmer that was tired of performing manually this refactoring to his code. Having lists of characters in clause patterns can be quite frequent in certain applications. However these lists of characters can be expressed much more elegantly by using strings instead. An example of the refactoring that will be performed by tidier is illustrated bellow (the example is fictitious):

```
foo(Str) ->
    case Str of
      "" -> ok;
      [$b, $a, $r | T] -> foo(T)
    end.
```

$\Downarrow$

```
foo(Str) ->
    case Str of
      "" -> ok;
      "bar" ++ T -> foo(T)
    end.
```

Although the refactoring is in itself trivial, the reader might agree that the new code is better looking and that the transformation is very tiresome to perform manually. Enhancing tidier with such a refactoring is a matter of minutes, while partially performing this transformation by hand might actually take more time. Moreover, adding such a refactoring to an automatic tool has one more advantage. From now on we have the ability to apply this transformation to any code (newer, older or of others) instantly, avoiding the time-consuming search-and-transform.

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 real 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.

Semantic equivalence All the aforementioned *record transformations* have no effect on the operational semantics of the program.

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.

4.3.1 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],
...
```

4.3.2 Eliminating `lists:keysearch/3`

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, Erlang/OTP 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 used to perform based on type information about the return values of the two functions (the use of the past tense will be explained later in this section). 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),
...
```

However, a couple of observations allowed us to fine-tune this refactoring and add one more useful transformation. After getting feedback from Erlang developers it became apparent that not everyone will find this transformation useful. The reason for that was the fact that many developers want their applications to be backwards compatible with earlier releases of Erlang/OTP and therefore were not keen to upgrade their code. Moreover, we observed that cases like the above, where the transformation is not immediate or cannot be performed, occurred very frequently in code. However, one can easily observe that in both examples the `lists:keysearch/3` function is used to check if a specific element is a member of a list. The Erlang `lists` module provides an easier and simpler way to perform such a check, the `lists:keymember/3` function. So the elimination of the `lists:keysearch/3` uses was split in two transformations: (i) one that simplified the search to a member-check and (ii) one that replaced `lists:keysearch/3` with `lists:keyfind/3`. They are activated by different flags so that a user can perform the `lists:keymember/3` transformation and at the same time retain backwards compatibility.

So the refactoring that will be suggested by tidier for the first of the above examples is shown below:

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

The `lists:keymember/3` function also allows us to transform the aforementioned assertion example which now becomes:

```
...
true = lists:keymember(delete, 1, Query),
...
```

Semantic equivalence The careful reader will notice that this transformation is only possible when the matching expression does not represent the value of another parent expression (for example the return value of a function). If this was not the case, the above transformation would change the value of the parent expression (from {`'value', tuple()`} to `'true'`) and therefore would not be semantics preserving.

This transformation came as a result of a 3-minute related talk with Erlang developers. We expect that a public release of the tool will result in a long wish-list and perhaps many more refactorings.
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.

4.3.3 Eliminating recursion

It must have become obvious by now that Erlang offers a variety of ways to express the programmer's intention. Finding the easiest and simplest way is never an easy task. Using recursion to perform list processing is an everyday routine for programmers that use functional languages. However, because of this habit programmers might miss opportunities to write a simpler and more elegant version of their code.

The *eliminating recursion* refactoring aims at the simplification of boolean functions that perform list processing by using the library functions `lists:any/2` and `lists:all/2` instead of recursion. In order to perform this transformation, tidier detects functions that emulate the `lists:[all,any]/2` function and replaces the body of these functions with a call to `lists:[all,any]/2`. This is currently one of the few complete function refactorings that are performed by tidier. We demonstrate the transformation by using an example from Erlang/OTP R13B's `lib/ic/src/ic_pragma.erl:1225` (slightly modified) :

```
is_decimal_str([]) -> true;
is_decimal_str([F | Rest]) ->
   case is_decimal_char(F) of
     true -> is_decimal_str(Rest);
     false -> false
   end.
```

$$\Downarrow$$

```
is_decimal_str(Fs) -> lists:all(fun is_decimal_char/1, Fs).
```

4.3.4 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 *lambda lifting*) refactoring we have discussed in Section 4.1.4 and also shown in Figure 4.1. 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.

4.3.5 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.

4.4.1 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))
```

$\Downarrow$

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

4.4.2 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)
```

$$\Downarrow$$

```
[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)
```

$$\Downarrow$$

```
[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)
```

$$\not\Downarrow$$

```
[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.

4.4.3 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).
```

$\Downarrow$

```
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);
```

$\Downarrow$

```
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.

Semantic equivalence The false branch of the fun expression has to be a match-all clause, so that the fun expression can be equivalent to the filtering effect of the list comprehension generator.

4.5 Transformations requiring type information

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

4.5.1 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 `==`

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

$\Downarrow$

```
foo(Rec, Fields, Key)
  when tuple_size(Rec)-1 =:= length(Fields) ->
    lists:zip([Key|Fields], tuple_to_list(Rec)).
```

Figure 4.2: A guard simplification refactoring from the actual code of CouchDB (`src/mochiweb/mochiweb_util.erl:422`).

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.

Semantic equivalence Tidier employs local type analysis to find opportunities for transforming coercing equalities and inequations with an integer to their matching counterparts. The compared have to be both integers in order for the transformation to be semantics-preserving. The analysis, although conservative, is often quite effective. The transformation itself is trivial. A conservative type analysis is the key element for semantics-preservation in all the refactorings that are performed by tidier and depend on type information.

4.5.2 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 [14], 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

```
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};
    ...
```

$\Downarrow$

```
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 4.3: Another guard simplification refactoring from actual code of Erlang/OTP (`lib/asn1/src/asn1rt_per_bin.erl:495`).

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.

4.5.3 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.

Two examples Figure 4.2 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 4.3 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.[1]

Semantic equivalence Since the redundant guards have to be in conjunction with the guards that imply their usage, the transformation is always semantics-preserving.

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.

4.6.1 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 (or macros) 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

[1]Actually, a global type analysis would discover that the `is_integer/1` guard is completely redundant in the code of Figure 4.3.

by hand on their programs.[2] 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.
```

$\not\Rightarrow$

```
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.

Semantic equivalence The structures that are assigned to a variable and reused have to be side-effect free. Moreover, due to the fact that tidier performs the transformations on source code containing macros (the preprocessor is not invoked), the structures must not contain or be within a macro context.

4.6.2 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

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

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 4.8). 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.

```
...
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.
```

$\Downarrow$

```
...
{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 4.4: An example of `case` straightening on actual code (from Erlang/OTP R13B's `lib/kernel/src/erl_boot_server.erl:274`).

4.7 Simplifying control

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

4.7.1 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 4.4. 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.

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.

Semantic equivalence This transformation is not *strictly* semantics-preserving. A *case clause* exception in the initial code will become a *bad match* exception in the refactored one. However, no exception will be silenced or missed by using this transformation. Therefore, according to the design goals we have set for tidier 3, we accept this refactoring as semantics-preserving. What is more, we should mention that, when the result of the case expression is assigned to a term, in order for the transformation to be correct, the straightened *case* expression has to be placed within a block expression and (if possible) to be flattened.

4.7.2 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.
```

⇒

```
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`.

```
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 4.5: Simplification of nested `case` expressions.

Switching on `true` and `false` is very common and this programming idiom often clutters the code unnecessarily. The clause on Figure 4.5 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 was taken from real code (`lib/percept/src/egd_render.erl:313`).

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

$\Longrightarrow$

```
Yp =:= Y
```

As another example, Figure 4.6 shows a significant simplification of Erlang Web's `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 [20] 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. 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.

4.8 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

```
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]).
```

$$\Downarrow$$

```
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]).
```

$$\vdots$$

$$\Downarrow$$

```
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 4.6: A case of multiple `if` simplifications.

following code:

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

By combining 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]
```

It appears that the above pattern is quite commonly employed for `lists:filter/2`. For example, the code of `lib/percept/src/percept_db.erl:394` which in Erlang/OTP R13B reads:

```
lists:filter(fun (Element) ->
                case Element of
                  {_, _, _} -> true;
                  _ -> false
                end
             end,
             ATs ++ STs ++ ITs)
```

is automatically transformed by tidier to:

```
[Element || Element = {_, _, _} <- ATs ++ STs ++ ITs]
```

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.

4.8.1 Inlining simple mapping funs

In Section 4.4.2 we introduced a refactoring that allows us to inline simple functions in list comprehensions and in 4.4.3 we explored a way to widen the field of filtering functions that could be inlined. However, something was missing. Mapping funs could also receive special treatment by tidier. The *inlining simple mapping funs* refactoring addresses single-clause mapping funs that do not have a match-all clause pattern.

In order to perform such a transformation, tidier has to ensure that every element of the generator-list will always match the clause pattern and therefore no function-clause exception can occur. Tidier handles this issue by performing a local type analysis, that guarantees that the generated patterns will always match the clause pattern[3]. A global type information analysis could certainly help tidier identify many more opportunities for such a refactoring (this is currently future work). However, it seems that even a simple local type analysis allows tidier to locate quite a lot of cases where this refactoring can be applied.

Tidier completes the transformation by merging the clause and generator patterns into a new generator pattern, and finally inlining the clause body into the list comprehension.

This refactoring can be especially effective when combined with previous transformations and the code being transformed has calls to `lists:map/2` and `lists:filter/2` nested within each other. As we will see, in some cases tidier is able to effectively perform *deforestation* [25] 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.

In figure 4.8.1 we show hand in hand the transformations that are automatically performed by tidier in a real code example (the code fragment is from Erlang/OTP R13B's `lib/inviso/src/inviso_tool_sh.erl:1638`). The combination of the various tidier refactorings leads to significant code simplifications and improvements.

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 similar case where deforestation can also be performed also occurs in the code of Wrangler (`src/refac_rename_fun.erl:344`):

[3]Specifically, tidier checks once more the syntactic equality of the generator and clause patterns

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

`lists:filter/2` to ⇓ list comprehension

```
get_all_tracing_nodes_rtstates(RTStates) ->
  lists:map(fun ({N,_,_}) -> N end,
            [X || X <- RTStates,
                        fun ({_,{tracing,_},_}) -> true;
                            (_) -> false
                        end(X)]).
```

inlining ⇓ filtering fun

```
get_all_tracing_nodes_rtstates(RTStates) ->
  lists:map(fun ({N,_,_}) -> N end,
            [X || X = {_,{tracing,_},_} <- RTStates]).
```

`lists:map/2` to ⇓ list comprehension

```
get_all_tracing_nodes_rtstates(RTStates) ->
  [fun ({N,_,_}) -> N end(V) ||
          V <- [X || X = {_,{tracing,_},_} <- RTStates]].
```

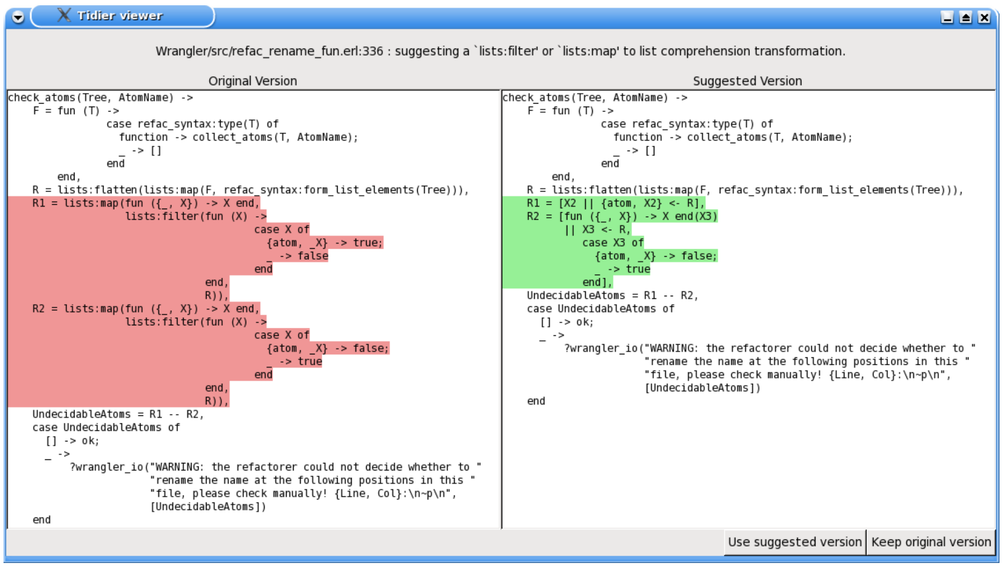
performing ⇓ deforestation

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

inlining ⇓ mapping fun

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

Figure 4.7: Showing the refactorings step-by-step, as they are applied by tidier to the code of `lib/inviso/src/inviso_tool_sh.erl:1638`



Wrangler/src/refac_rename_fun.erl:336 : suggesting a `lists:filter' or `lists:map' to list comprehension transformation.

Original Version

```
check_atoms(Tree, AtomName) ->
    F = fun (T) ->
                case refac_syntax:type(T) of
                  function -> collect_atoms(T, AtomName);
                  _ -> []
                end
        end,
    R = lists:flatten(lists:map(F, refac_syntax:form_list_elements(Tree))),
    R1 = lists:map(fun ({_, X}) -> X end,
                   lists:filter(fun (X) ->
                                        case X of
                                          {atom, _X} -> true;
                                          _ -> false
                                        end
                                end,
                                R)),
    R2 = lists:map(fun ({_, X}) -> X end,
                   lists:filter(fun (X) ->
                                        case X of
                                          {atom, _X} -> false;
                                          _ -> true
                                        end
                                end,
                                R)),
    UndecidableAtoms = R1 -- R2,
    case UndecidableAtoms of
      [] -> ok;
      _ ->
          ?wrangler_io("WARNING: the refactorer could not decide whether to "
                       "rename the name at the following positions in this "
                       "file, please check manually! {Line, Col}:\n~p\n",
                       [UndecidableAtoms])
    end
```

Suggested Version

```
check_atoms(Tree, AtomName) ->
    F = fun (T) ->
                case refac_syntax:type(T) of
                  function -> collect_atoms(T, AtomName);
                  _ -> []
                end
        end,
    R = lists:flatten(lists:map(F, refac_syntax:form_list_elements(Tree))),
    R1 = [X2 || {atom, X2} <- R],
    R2 = [fun ({_, X}) -> X end(X3)
          || X3 <- R,
             case X3 of
               {atom, _X} -> false;
               _ -> true
             end],
    UndecidableAtoms = R1 -- R2,
    case UndecidableAtoms of
      [] -> ok;
      _ ->
          ?wrangler_io("WARNING: the refactorer could not decide whether to "
                       "rename the name at the following positions in this "
                       "file, please check manually! {Line, Col}:\n~p\n",
                       [UndecidableAtoms])
    end
```

Figure 4.8: Tidier simplifying the code of Wrangler.

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

Tidier automatically 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. In figure 4.8 we can see tidier performing the aforementioned transformation

Semantic equivalence In order for the *deforestation* to be semantics-preserving, the mapping and filtering funs must be side-effect free. Furthermore, we should mention that this transformation is also *not strictly* semantics-preserving. Although no exceptions will be missed or silenced, performing this refactoring might change the sequence in which the exceptions will occur.

4.8.2 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 [19]) 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.

4.9 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 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 real code excerpt which is taken 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 4.9. 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.

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

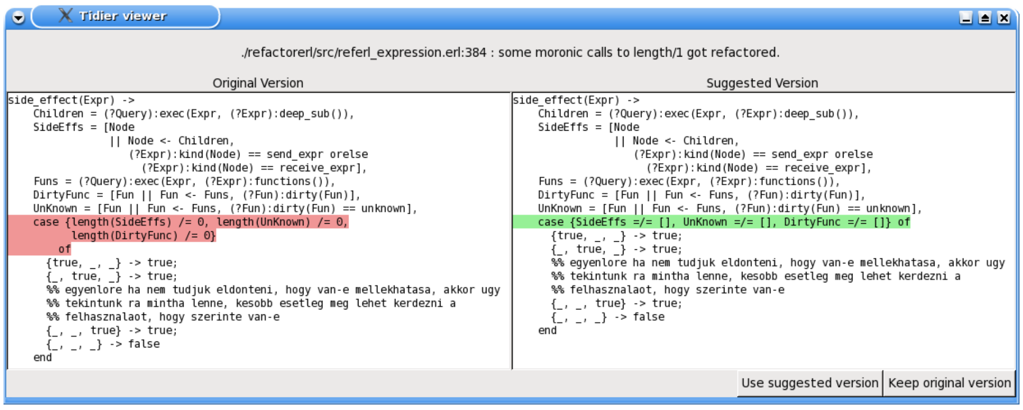
Tidier has also located a clause with three unnecessary calls to `length/1` next to each other. The code is from the latest released version of RefactorErl. Its refactoring is shown in Figure 4.10. Neither we nor tidier understand the comment in Hungarian, but we are pretty sure that the whole `case` statement can be written more simply as:

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

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

Once this refactoring was in place, we observed that we could eliminate even more uses of length (especially when they were used to check whether a list has a small number of elements) by using a matching. What is more, we could now introduce variables for the elements of the matched list and use them to extinguish calls to `hd/1` and `lists:nth/2` that traversed the matched list. There were actual cases where the initial list variable was not even needed (as we can witness in figure 4.11).

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 transformed the *if expression* to a *case expression* and eliminated the call to `hd/1` as part of the transformation.



./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 4.10: Tidier simplifying the code of RefactorErl `src/referl_expression.erl`.

```
process_body(Args) when length(Args) == 2 ->
    Pid = element(2, lists:nth(1, Args)),
    Node = element(2, lists:nth(2, Args)),
    "<BODY BGCOLOR=\"#FFFFFF\">" ++ m:foo(Pid, Node) ++ "</BODY>".
```

$\Downarrow$

```
process_body([Args1, Args2]) ->
    Pid = element(2, Args1),
    Node = element(2, Args2),
    "<BODY BGCOLOR=\"#FFFFFF\">" ++ m:foo(Pid, Node) ++ "</BODY>".
```

Figure 4.11: Length simplification from R13's `lib/appmon/src/appmon_web.erl:647`

```
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`):

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

Tidier will transform immediately the above code excerpt to the following:

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

Very recently we have added one more functionality to our *length removal* refactorings. After reviewing a lot of code, we discovered a very common idiom among Erlang programs: a large percentage of the occurrences of `lists:duplicate/2` were used in conjunction with an application of `length/1` to another list in order to calculate the length of the duplicated list. However, the call to `length/1` is completely unnecessary since it can be avoided by using a list comprehension. We use the following example from Erlang/OTP R13B's `lib/compiler/src/cerl_clauses.erl:335` to illustrate the refactoring:

```
...
Es = lists:duplicate(length(Ps), any),
...
```

$$\Downarrow$$

```
...
Es = [any || _ <- Ps],
...
```

The refactored version is not only shorter and simpler in terms of source code size and quality; but also probably executes faster since it there is no need to calculate the length of the list and the duplicated list is created in one traversal.

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

Semantic equivalence Although these refactorings are quite simple, one should be once more very careful with the removal of `length/1`. In order for the transformation to be semantics-preserving (and not to miss any exceptions) tidier has to make certain that no exceptions will occur while applying `length/1` to its argument. There are two ways for tidier to verify this: (i) if the application of `length/1` is in guard context or (ii) if the argument can be proven to be a list via type analysis.

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.

Chapter 5

Experience

5.1 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
```

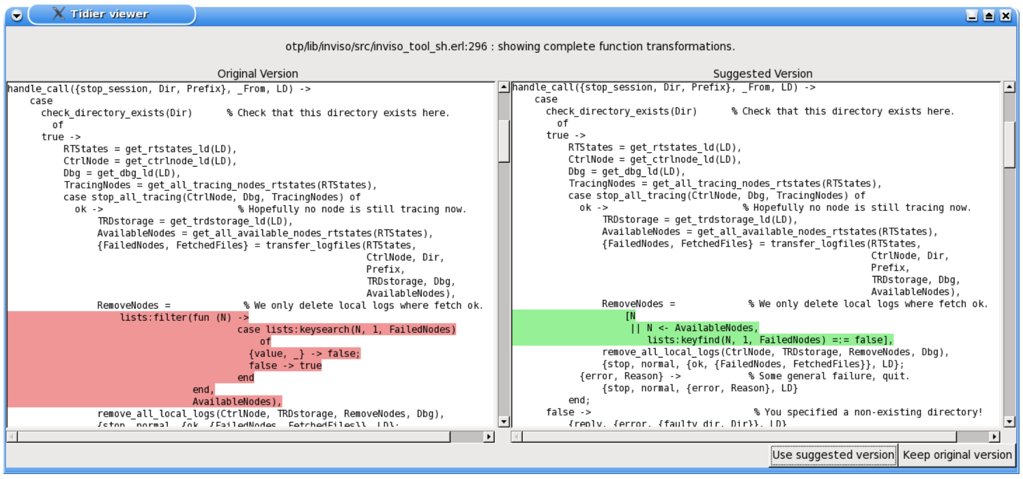
which will tidy all `*.erl` files under these directories. Both of these commands will apply the default set of transformations on all files. If only some of the transformations are desired, the user can select them via appropriate command-line options. For example, one can issue the command:

```
> 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 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 5.1 shows tidier in action. In fact, the snapshot depicts tidier refactoring a file from the `inviso` application of Erlang/OTP R13B.



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 5.1: Tidier in action: simplifying the source code of 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.

5.1.1 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 follows in the next section. 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.

5.2 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 [4]. 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 [5]. We used version 0.2 of Disco. Its core is written in Erlang and consists of about 2,500 lines of 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.

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 [6]. 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 [7]. 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 [15]. 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 [23]. 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 [27] is also included as a component.

Wings 3D is a subdivision modeler for three-dimensional objects [26]. 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 [12] 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.

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 5.1. 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 5.1 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 5.2 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.)

	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 5.1: Number of tidier's transformations on various Erlang source code bases.

	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 5.2: Effectiveness of tidier' refactorings requiring type info.

5.3 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.

Chapter 6

Related Work

6.1 Related Work

Software refactoring [8], 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 [17] 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 [13], the ViPReSS tool for Prolog [24], 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 [16] or refactor the module structure of an existing application by using code clustering techniques [15]. 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 [12] and quite recently has also been enhanced with the ability to detect and remove duplicated code [11]. All these refactorings are initiated and controlled by the programmer. According to a published survey of Erlang tools [18] 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 [21] add-in to Visual Studio, we were not able to locate any other fully automatic code cleanup tools in any high-level language.[1] We hope that tidier will pave the way for more fully automatic code simplification and cleanup tools in Erlang and other languages.

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

Chapter 7

Conclusion and Acknowledgements

7.1 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. A subset of the refactorings that were described are general enough to be applied as is or with minor changes in other functional programming languages. 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. 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 simply or elegantly. 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.

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.

7.2 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. We also thank Björn Gustavsson, and Kenneth Lundin for supportive comments and suggestions for refactorings. Finally we would like to 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.

Last but not least, 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.

Bibliography

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

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

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

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

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

[9] 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.

[10] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Functional Programming Languages and Computer Architecture*, pages 190–203. Springer-Verlag, 1985.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

Appendix A

Tidier Reference Manual

Tidier is a static analysis software refactoring tool that can automatically perform transformations that improve the quality of Erlang programs.

A.1 Tidier

A.1.1 Introduction

Tidier is a static analysis software refactoring tool that can automatically perform transformations that improve the quality of Erlang programs. To do so, it implements a suite of bad code smell detectors (that e.g. find uses of obsolete language constructs, unnecessarily long and complicated expressions, obfuscated ways of expressing the programmers' intention, etc.) and rewrites these code pieces with semantically equivalent, simpler and more modern language expressions.

Tidier was implemented in order to help the programmer track down potential problems in his code, acting as an automatic software refactoring tool. However, Tidier has an option which does not allow the modification of the source code of the actual files. This option is very handy and allows tidier to be used as a bad code smell detector. The user can also handle tidier from the GUI interface in both modes (refactorer and bad code smell detector). By enabling the GUI interface, the programmer can see the refactorings that are suggested or performed by Tidier and interactively accept or reject the transformations. For a complete list of the supported options and their uses see A.2.1.

A.1.2 Supported Transformations

Tidier can suggest a variety of source code transformations. These transformations are listed and described in detail in chapter 4.

A.2 Using Tidier from the command line

Tidier can be used through the command line via the tidier c-frontend that has been built for automated use. In the following section we will present a list of the supported flags and the transformations that are activated by each one.

A.2.1 Options

The following information regarding the usage and options of Tidier can be obtained by writing 'tidier --help' in a standard shell.

Usage

tidier [--help] [--version] [--gui] [--no-transform]
[--all] [--any] [--apply] [--boolean] [--case-simplify] [--catch]
[--comprehensions] [--exact] [--experimental] [--fun-expressions]
[--guards] [--imports] [--intermediate] [--lists] [--records]
[--patterns] [--r13] [--size] [--straighten] [--structs]
[--quiet] [--verbose] file* |dir*[2]

Main switches

--gui (or -g) (requires the presence of wxWidgets) opens a GUI interface to show the code transformations and allow changes to them by the programmer

--no-transform when specified, no changes will be applied to the input files

Transformation options

--all Performs (almost) all the transformations mentioned below.

--any Simplifies boolean recursive functions that process lists by employing 'lists:any/2' and 'lists:all/2' (4.3.3).

--apply Changes uses of 'apply/2' and 'apply/3' to direct function calls (4.1.3).

--boolean Simplifies boolean expressions (4.7.2).

--case-simplify Finds opportunities to simplify case and if expressions with boolean clauses (4.7.2). Might also employ 'erlang:min/2' and 'erlang:max/2' (4.1.5).

[0]The '*' denotes that multiple occurrences of these options are possible

--comprehensions Finds opportunities for using list comprehensions instead of calling 'lists:map/2' and 'lists:filter/2'. In addition, it simplifies list comprehensions and performs deforestation whenever possible (4.3, 4.4 and 4.8).

--exact Transforms coercing to exact arithmetic equalities and inequations (4.5.1).

--experimental unnecessary calls to 'length/1' (usually used as guards) will be rewritten using simpler but equivalent expressions (4.9).

--fun-expressions Transforms fun epressions to local function calls (lambda lifting - 4.1.4).

--guards Finds opportunities for transforming old style guards (e.g. atom(X)) to new style guards (is_atom(X)) and for simplifying guard expressions (4.1.1).

--imports All import statements will be removed and calls to imported functions will be expanded to explicit remote calls (4.1.2).

--intermediate Performs temporary variable elimination if this leads to significant simplifications (4.6.2).

--lists Calls to 'lists:append/2' and 'lists:subtract/2' will be rewritten using the '++' and '--' operators and calls to 'lists:keysearch/3' will be rewritten (whenever possible) as calls to 'lists:keymember/3' (4.3)

--records Finds opportunities for transforming is_record/1 guards to pattern matching in function clauses and case statements and for moving record accesses to the place where the record is deconstructed (4.2).

--patterns Converts patterns that contain lists of characters to strings (4.1.6).

--r13 Replaces occurrences of lists:keysearch/3 with 'lists:keyfind/3'. It is usually a good idea to activate this flag in conjunction with '--lists'. Should be used with Erlang/OTP R13 or higher (4.3).

--size Changes calls to 'size/1' to 'tuple_size/1' or 'byte_size/1'; in doing so, it also eliminates unnecessary uses of 'is_tuple/1' or 'is_binary/1' (4.5.2 and 4.5.3).

--straighten Finds opportunities to simplify case expressions with a single alternative clause and create straight-line code (4.7.1).

--structs Finds opportunities to reuse already created structures instead of reconstructing them (4.6.1).

Miscellaneous

--quiet information messages and warning messages will be suppressed

--verbose progress messages will be output while the program is running

--help (or -h) prints this message and exits

--version (or -v) prints the version number and exits