Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Στατική Ανάλυση της Αγνότητας των Συναρτήσεων της Γλώσσας Erlang

Διπλωματική Εργασία

του

Μιχάλη Πιτίδη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εργαστήριο Τεχνολογίας Λογισμικού
Αθήνα, Ιούλιος 2010

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Στατική Ανάλυση της Αγνότητας των Συναρτήσεων της Γλώσσας Erlang

Διπλωματική Εργασία

του

Μιχάλη Πιτίδη

Επιβλέπων: Κωστής Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 16η Ιουλίου, 2010.

....	....	....
Κωστής Σαγώνας	Νικόλαος Παπασπύρου	Κώστας Κοντογιάννης
Αν. Καθηγητής Ε.Μ.Π.	Επικ. Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2010

....
Μιχάλης Πιτίδης
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
© 2010– All rights reserved

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών
Εργαστήριο Τεχνολογίας Λογισμικού

Copyright © – All rights reserved Μιχάλης Πιτίδης, 2010.
Με επιφύλαξη παντός δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Περίληψη

Οι αγνές συναρτήσεις, δηλαδή οι συναρτήσεις χωρίς παρενέργειες, παίζουν σημαντικό ρόλο στις συναρτησιακές γλώσσες προγραμματισμού, καθώς βοηθούν στη συγγραφή κατανοητού κώδικα που είναι εύκολο να ελεγχθεί. Σε μια γλώσσα σαν την Erlang, η οποία δεν διαθέτει σύστημα τύπων και επιτρέπει την αδιάκριτη χρήση αγνών και μη αγνών συναρτήσεων, η διαβεβαίωση ότι ορισμένες συναρτήσεις είναι αγνές μπορεί να φανεί χρήσιμη σε συγκεκριμένες περιπτώσεις. Η παρούσα διπλωματική εργασία επικεντρώνεται στην υλοποίηση μιας πλήρως αυτοματοποιημένης στατικής ανάλυσης για τη διαπίστωση της αγνότητας ή μη των συναρτήσεων Erlang. Καταγράφει τις απαραίτητες ιδιότητες που καθιστούν μια συνάρτηση αγνή και περιγράφει τη σχεδίαση της εφαρμογής Purity, ενός εργαλείου που ξεκινάει από ένα σύνολο συναρτήσεων με προκαθορισμένη αγνότητα για να αναλύσει τον κώδικα που του παρέχει ο χρήστης. Με μια κατά βάση απλή και συντηρητική προσέγγιση, είναι δυνατό να αποφανθούμε για την αγνότητα ή μη του σχεδόν 90% των συναρτήσεων στις εφαρμογές που εξετάζουμε.

Σαν πρακτική εφαρμογή, η ανάλυση μας ενσωματώθηκε στο μεταγλωττιστή της Erlang, επιτρέποντας τη χρήση αυθαίρετων αγνών συναρτήσεων σε εκφράσεις φρουρών, κάτι που προηγουμένως δεν ήταν δυνατόν. Η προσπάθεια αυτή δεν ήταν πλήρης, καθώς στόχος της ήταν να δείξει ότι κάτι τέτοιο είναι εφικτό και ενδεχομένως να αποτελέσει κίνητρο για μια ωριμότερη υλοποίηση. Πέρα από επεκτάσεις σαν την προηγούμενη, τα αποτελέσματα της ανάλυσης μας θα μπορούσαν να επιτρέψουν ορισμένους τύπους βελτιστοποιήσεων στον μεταγλωττιστή της Erlang.

Λέξεις Κλειδιά

στατική ανάλυση, αγνά συναρτησιακός προγραμματισμός, ανάλυση αγνότητας συναρτήσεων, παρενέργειες, Erlang

Abstract

Pure functions play an important role in functional programming languages, and help in writing easy to test, robust and comprehensible code. In a language like ERLANG, which lacks a type system and allows pure functions to be used interchangeably with impure ones, being able to reason about the purity of certain functions can prove useful. This thesis focuses on the implementation of a fully automatic static analysis that determines the purity of ERLANG functions. It identifies the necessary properties of pure functions, and describes the design of PURITY, a tool which builds upon a set of functions with predetermined values to analyse the purity of user provided code. Based on a generally simple and conservative approach, it was possible to conclusively determine the purity of roughly 90% of the functions in the code bases we tested.

As a practical application, our analysis was integrated into the ERLANG compiler, allowing arbitrary pure functions to be used in guard expressions, something not previously possible in ERLANG. While a bit rough and incomplete, our proof of concept could provide motivation for a more mature implemenation. Furthermore, our analysis could make way for some types of optimisations in the ERLANG compiler.

Keywords

static analysis, purely functional programming, pureness analysis, side-effects, Erlang

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή μου, Κωστή Σαγώνα, για την ευκαιρία που μου προσέφερε και για την συμβολή του στην εκπόνηση της συγκεκριμένης εργασίας. Η καθοδήγηση του στα τελευταία έτη των σπουδών μου ήταν καθοριστική.

Θέλω να ευχαριστήσω επίσης όσους φίλους και συμφοιτητές μου στάθηκαν κατά την παραμονή μου στη σχολή.

Τέλος, ευχαριστώ θερμά την οικογένεια μου για τη στήριξη και την υπομονή τους όλα αυτά τα χρόνια. Πάνω απ' όλα όμως, ευχαριστώ τη Γεωργία Φωτοπούλου, που με την πολυτιμότατη και αδιάκοπη στήριξη της έθεσε τις προϋποθέσεις για την διεκπεραίωση της εργασίας αυτής, και όχι μόνο.

Μιχάλης Πιτίδης

Contents

Περίληψη 1

Abstract 3

Ευχαριστίες 5

Contents 8

List of Figures 9

List of Tables 11

Listings 13

1 Introduction 15
- 1.1 Motivation and Outline of the Thesis . . . 15
- 1.2 Purely Functional Programming . . . 15

2 Preliminaries 17
- 2.1 Erlang . . . 17
  - 2.1.1 Implementation details . . . 17
  - 2.1.2 Built-in functions . . . 17
  - 2.1.3 Data types . . . 17
  - 2.1.4 Exceptions . . . 18
  - 2.1.5 Pattern Matching: En Guard! . . . 18
  - 2.1.6 Concurrency . . . 20
  - 2.1.7 On-the-fly code reloading . . . 21
  - 2.1.8 Erlang NIFs . . . 21
- 2.2 Core Erlang . . . 21
- 2.3 What is Pure in Erlang . . . 22
  - 2.3.1 Except exceptions . . . 22
  - 2.3.2 Examples . . . 23

3 Analysis 25
- 3.1 Description . . . 25
  - 3.1.1 Bootstraping the analysis . . . 27
  - 3.1.2 PLT: the Persistent Lookup Table . . . 27
  - 3.1.3 Mutually recursive functions . . . 28
- 3.2 Higher Order Functions and Limitations . . . 29

3.3 A Conservative Approach . . . 32
3.4 Purity in Numbers . . . 32
3.4.1 Beyond the numbers . . . 36

4 User Defined Guards **39**
4.1 Motivation . . . 39
4.2 Erlang Compilation . . . 41
4.3 Implementation . . . 41
4.3.1 Frontend . . . 41
4.3.2 Backend . . . 41
4.4 Engineering Issues . . . 41
4.5 Usage Examples . . . 42
4.6 Limitations . . . 44

5 Related Work **45**
5.1 Purity in other Languages . . . 45
5.1.1 The case of Haskell . . . 45
5.1.2 Clean . . . 46
5.1.3 BitC . . . 46
5.1.4 Joe-E . . . 46
5.2 Other Uses . . . 46
5.2.1 Compiler optimizations . . . 46
5.2.1.1 Common Subexpression Elimination (CSE) . . . 46
5.2.1.2 Code Reordering . . . 46
5.2.1.3 Parallelization . . . 47
5.2.1.4 Memoization . . . 47
5.2.2 Miscellaneous . . . 47

6 Conclusion **49**
6.1 Concluding Remarks . . . 49
6.2 Future Work . . . 49
6.3 Acknowledgements . . . 49

Bibliography **50**

List of Figures

3.1 Percentage of pure functions in OTP and ibrowse 35
3.2 Percentage of pure functions in Erlsom and Yaws 35
3.3 Percentage of pure functions in ejabberd and Wings3D 36
3.4 Percentage of pure functions in CouchDB and purity 36

List of Tables

2.1 Common impure constructs in Erlang 23

3.1 Details of analysed applications 34
3.2 Analysis results 34

Listings

1.1 Pure functions vs functions executed for their side-effects 16
2.1 A simple example of pattern matching 19
2.2 The previous example in a language without pattern matching 19
2.3 Example of pattern matching with guards 20
2.4 Exceptions in guards are silenced 21
2.5 Exceptions can be tricky 22
3.1 A genuinely pure function 25
3.2 Pure functions which may raise an exception 26
3.3 Part of Listing 3.2 translated to Core Erlang 27
3.4 A stateful accumulator 27
3.5 Concurrency is a side-effect 28
3.6 Examples of mutually recursive functions 29
3.7 An example of a higher order function 29
3.8 A more elaborate example of higher order functions 30
3.9 Mutually recursive higher order functions 31
3.10 Functions as parts of complex data structures 31
3.11 Conditional higher order functions 32
3.12 Functions as return values 32
3.13 Example of limitations on complex variable assignments 32
3.14 Is this function impure? 37
4.1 Compact guards with the help of macros 39
4.2 Employing custom abstract type tests in guards 40
4.3 More cases of useful user defined guards 40
4.4 A very basic user defined guard 42
4.5 Example usage of the modified compiler 42
4.6 A more useful guard 43
4.7 Different results depending on the presence of the PLT 43
4.8 Verifying the guards work as expected 43
4.9 Limitations on direct calls to higher order functions 44
5.1 Encapsulation of impure functions into monads 45

Chapter 1

Introduction

1.1 Motivation and Outline of the Thesis

The purpose of this thesis is to design and implement a lightweight static analysis for determining the purity of ERLANG functions, and evaluate the effectiveness of such an analysis in actual code bases.

One may wonder why such an analysis is of interest. To start with, this analysis can be useful to other software development tools. Compiler writers might benefit from such information and enable certain optimisations based on it. Other type of code transformation tools, such as automatic refactoring programs, might find this analysis useful as well [3, sec. 5]. Furthermore, such a tool can be of use to language designers, to gain insight into what programs their users usually write. Programmers may also find it helpful in identifying good or bad coding practices in their code, and restructuring it.

Our primary goal is to use this analysis as a stepping stone for extending the ERLANG runtime and language supporting arbitrary guard expressions. While the first part of this thesis elaborates on the analysis and its implementation —an ERLANG application aptly named PURITY— the latter documents our experiences implementing a prototype of such an extension.

Before getting into the details of our implementation, we give an overview of purity in functional programming languages and examine why it may represent a desired property of functions. Chapter 2 is an introduction to the ERLANG programming language, focusing on those features which influence our analysis. Following that, a detailed description of the analysis itlself is described in Chapter 3. Chapter 4 details the process of implementing a small prototype for user defined guards in the ERLANG compiler, while Chapter 5 presents a brief summary of related work. Chapter 6 concludes this thesis.

1.2 Purely Functional Programming

Functional programming is a programming paradigm whereby computation is modeled as the evaluation of expressions. This is in stark contrast to imperative programming, which models computation as a series of statements (or "actions"), mutating the global state as the statements are executed.

A function is said to be pure, if its outcome is not dependent on any mutable internal state, which could vary across the program's execution, and if it does not modify its environment in any way. Such functions will always produce the same value given the same set of arguments, a property known as *referential transparency* [11].

Since evaluation of expressions and functions plays a central role in functional languages, the notion of a *side-effect* is introduced, i.e., any other interaction with the program's state or its environment. Characterising a function as having side-effects is synonymous to saying it is not pure.

Listing 1.1 illustrates the difference with a very simple example in pseudo-code. The first function, `add`, is useful because of the calculation it performs, and subsequently returns. Conversely, the `say_hello` function is only useful because of its side-effect, writing the message `"Hello, world!"` to standard output; its return value is irrelevant.

```
add x y =
  x + y

say_hello =
  writeln "Hello, world!"
```

Listing 1.1: Pure functions vs functions executed for their side-effects

A functional language which only allows pure functions is said to be purely functional. There are certain advantages to the purely functional approach in language design. First of all, reasoning about pure functions is easier, and more intuitive. The programmer does not have to keep track of all the possible changes in state, pertaining to a function call. Pure functions are closer to their mathematical counterparts.

Furthermore, there is room for some types of optimisations, only possible with referentially transparent functions. Order of evaluation is not important with pure functions, and a compiler is free to reorder expressions as necessary, to achieve better locality of reference, or parallelization. Multiple calls to a function with the same set of arguments can be substituted by a single one, and the result used in all subsequent calls, an optimisation known as *common subexpression elimination*.

However, there are certain drawbacks as well. Having mutable data types can be more efficient for a sizeable class of programs. Communication with the environment, be it I/O, message passing or something else, is inherently imperative and mostly procedural.

Traditionally, functional languages work around these problems by including a small set of imperative constructs, while keeping the core of the language purely functional. This is the approach taken by ERLANG. A notable exception is Haskell, which manages to keep the language purely functional, by wrapping side-effects and imperative constructs in monads [12].

Chapter 2

Preliminaries

2.1 Erlang

Erlang is a functional language, with strict semantics and dynamic types. Besides features commonly found in functional languages, such as automatic memory management, and built-in support for high level data types, some key features make Erlang stand out. These include first class support for concurrency and distribution, as well as fault-tolerance and on-the-fly code reloading. The language also provides soft real-time guarantees. The above make Erlang ideal for building highly scalable and robust systems.

While functional programming is an established paradigm in the academic world, it is not as widespread in the computer industry. Erlang is a notable exception in this respect, being one of the most industrially relevant functional languages of our day.

2.1.1 Implementation details

The de-facto implementation of the Erlang VM is the Bogdan/Björn's Erlang Abstract Machine, BEAM for short. Compilation of Erlang programs ends with generation of BEAM bytecode, which is then interpreted by the virtual machine —from here on abbreviated to VM.

2.1.2 Built-in functions

Built-in functions —or simply BIFs— are special functions in Erlang. They are native to the virtual machine and implemented in the language the VM is written, in this case C. As a consequence, their execution differs from the execution of regular Erlang functions. For example, the BEAM bytecode generated for a call to some built-in function is typically less complicated than that generated for regular function calls.

BIFs are thoroughly documented in the Erlang module man page of the ERTS reference manual [8]. The manual also includes information on which BIFs can be used in guard expressions. Why such information is necessary is the subject of Section 2.1.5.

2.1.3 Data types

A brief introduction to the most common data types in Erlang is presented in this section. These provide the building blocks for more complex structures. For a more thorough description, refer to the Erlang manual [7, ch. 2].

Numbers Erlang supports integers and floats, e.g., `1`, `42`, `3.14`.

Atoms These are essentially named constant literals, e.g., `foo`, `bar`, `ok`, `error`.

Lists A collection of ERLANG terms, which can be of arbitrary length and include terms of any type, e.g., `[1,3.14,foo]`.

Tuples A collection of ERLANG terms, this time of fixed length, e.g., `{1,3.14,foo}`.

Records Not a distinct data type, but rather a convenience notation for referring to tuple elements by name.

Bit Strings and Binaries These are used to represent areas of untyped memory and allow for efficient access to parts of it through a special syntax, e.g. `<<10,20>>`, `<<"ABC">>`

Besides the ones mentioned above, ERLANG supports some additional built-in types, such as port and process identifiers, or references, which are implementation dependent. It should also be noted that strings and booleans are a special case of the previous types. Strings are lists of characters, where characters are represented by integers, while boolean values are the atoms `true` and `false` respectively.

2.1.4 Exceptions

ERLANG has a relatively simple exception handling mechanism. An exception consists of its class, an exit reason, and a stack trace, which are ERLANG terms themselves. Exceptions of class `error` are generated by runtime errors, which occur when builtins such as pattern matching, arithmetic and list manipulation operations fail. The user may also generate an exception with one of the `exit/1` and `throw/1` BIFs, which belong to an equivalent class.

If uncaught, an exception will terminate the process which evaluated the erroneous expression. Two distinct mechanisms may catch exceptions, the older `catch` expression, or the newer and more robust `try-catch`, which can distinguish between different exception classes. The various exit reasons are documented in the ERLANG manual [7, ch. 10].

2.1.5 Pattern Matching: En Guard!

A well established feature of many functional languages, particularly those influenced by ML, *pattern matching*, plays a central role in ERLANG. Pattern matching tries to match a sequence of values against a corresponding sequence of patterns. The result, if successful, is a mapping of variables from the pattern, to the various terms in the sequence of values.

One can think of pattern matching as the opposite of constructing a value. Whereas in construction we bind a variable to a concrete value we specify, in pattern matching we try to match an abstract value —such as a function argument— to a concrete one, deconstructing it and binding variables to different parts of it. For instance, one can assign a single-element list to a variable, with something like `L = [5]`. To pattern match on such a list, binding the element to a variable, one merely has to write `[E] = L`.

While pattern matches in ERLANG can occur as arbitrary expressions in function bodies, like the one just shown, they are most useful if specified in function definitions, `case`, `receive` and `try-catch` expressions. In such cases, multiple patterns can be tried sequentially, until a matching one is found.

Besides lists, most combinations of ERLANG data types can be pattern matched. This includes numbers, atoms, tuples and records. Pattern matching also plays a major part in

the concurrent aspect of the language, as it used extensively in the extraction of messages from the mailbox of a process with the `receive` expression.

The example in Listing 2.1 shows how pattern matching can help write concise and declarative functions. This particular example implements a lookup function for association lists, which is very close to the abstract definition of such an operation:

1. if the list is empty, the key is not included;
2. if the first element of the list matches the key being looked up, stop and return the value; or
3. if the first element of the list does not match the key, search the remainder of the list.

Listing 2.2 contrasts a similar implementation in a fictional language without pattern matching. This version is slightly convoluted, and does not convey the semantics of the function so clearly.

Note how we can use the same name in two or more variables and the match will only be successful when they hold the same value. Also noteworthy is the use of variable names with a leading underscore. These are special to the compiler and will not generate a warning if they remain unused in the rest of the function. Another special variable, the single underscore, is not bound to values at all and can be used as a catch-all for any part of the pattern we do not care about.

```
1 assoc_find(_Key, []) ->
2     error;
3 assoc_find(Key, [{Key, Val}|_T]) ->
4     {ok, Val};
5 assoc_find(Key, [_H|T]) ->
6     assoc_find(Key, T).
```

Listing 2.1: A simple example of pattern matching

```
1 assoc_find(Key, Lst) ->
2   if Lst =:= []
3     error
4   else
5     H = hd(Lst),
6     if is_tuple(H) and size(H) =:= 2 and element(1, H) =:= Key
7       {ok, element(2, H)}
8     else
9       assoc_find(Key, tl(Lst))
```

Listing 2.2: The previous example in a language without pattern matching

When a pattern match is incomplete, and there is no matching clause, an exception will be raised. The reason varies between `badarg`, `function clause` and `case clause`, depending on the case. For example, the `length/1` function, which is not total, will raise a `badarg` exception when called with anything other than a list, as depicted in Listing 2.4.

Additional constraints can be placed on pattern matches, with the use of guards. Guard expressions are executed in order and if the result evaluates to true, then the match is successful. With guards, it is possible to extend the expressiveness of pattern matching,

adding support for things like value ranges for numbers, or tests for abstract types, like process identifiers, references and function objects. Listing 2.3 shows an example of pattern matching extended with guards.

```
cool(T) when T > 15, T < 22 ->
  true;
cool(T) ->
  false.

loop_register(#st{players = Ps} = St0, Waiting) ->
    receive
        {Pid, start} when Ps =/= [], is_pid(Pid), node() =:= node(Pid) ->
            %% Override player wait.
            loop_register(St0, 0);
        {new_player, Node, Pid, {Name, Vsn} = _Info} when is_pid(Pid) ->
        ...
   end.
```

Listing 2.3: Example of pattern matching with guards

Since guard expressions are executed for every pattern match candidate, it is evident that they should lack side-effects. Furthermore, guard expressions should evaluate in bounded time. ERLANG's solution is to restrict valid guard expressions to a predefined set, which includes variables, boolean and arithmetic operators, and a small set of pure BIFs, such as type tests.

A notable exception to the above are the `node/0` and `node/1` BIFs. Even though they lack side-effects, they depend on the environment, specifically the node name of the VM the code is currently executed in, which can be changed dynamically at runtime. This is a testament to the ad-hoc manner in which certain parts of the ERLANG VM and language specification have evolved.

Invalid guard expressions are detected at compile time, and the user is notified with an appropriate error message. This occurs early in the compilation process, so for the most part the compiler back-end works on the assumption that function calls in guard expressions can only be BIFs.

Guards can fail for two reasons. Either the expression evaluates to false, or it raises an exception. Consequently, any exception raised in a guard is silenced. This is apparent in Listing 2.4, where the call to `length(42)` raises a `badarg` exception, while the call to `Foo(42)` does not. Internally, the compiler wraps guard expressions inside `try-catch` blocks.

2.1.6 Concurrency

Erlang implements the *Actor* model of concurrency. In particular, its concurrency model can be summarised as *share-nothing* concurrency based on lightweight processes communicating via asynchronous message passing [2].

We can see this model as an extension to that of pure functions. Each process does not share state with other processes, ideally at least. Instead, it can only receive and send messages, much like a function can be thought of *receiving* its arguments and *sending* its reply.

This way it is far easier to reason about complex concurrency schemes. However, just like with functions, pragmatic constraints force ERLANG to allow for global struc-

```
1> Foo = fun(L) when length(L) =:= 4 -> ok;
1>            (_) -> error end.
#Fun<erl_eval.6.13229925>
2> length(42).
** exception error: bad argument
     in function  length/1
        called as length(42)
3> Foo([1,2,3,4]).
ok
4> Foo([1,2]).
error
5> Foo(42).
error
```

Listing 2.4: Exceptions in guards are silenced

tures, like the process registry, or ETS and DETS tables, which can introduce unwanted complications such as race conditions [6].

2.1.7 On-the-fly code reloading

An unusual feature of Erlang is support for replacement of old code with a newer version, while the system is still running, otherwise known as on-the-fly code reloading. This has implications on our plan to add support for user defined guards, since it is not enough to integrate it to the compiler; a check for pure functions —or some type of assurance thereof— has to be placed somewhere in the code loader of the Erlang runtime system as well.

2.1.8 Erlang NIFs

Native Implemented Functions (NIFs) are a relatively new Erlang construct, which allow interfacing Erlang with C code in a simple manner. The standard way of doing so would be through the Erlang ports system, which spawns external processes or links C code as a shared library, and uses message passing to communicate with an Erlang program. The old approach is clearly an impure one, for the concurrency part alone.

As stated in the NIF section of the Erlang interoperability tutorial [9], NIFs are meant as synchronous functions making relatively short calculations without side-effects. However, this is just a guideline and cannot be checked or enforced either way. Thus, the purity of NIFs has to be hard-coded, in a manner similar to BIFs, by the user of the application.

2.2 Core Erlang

As part of the compilation process, Erlang code is first translated to Core Erlang, a simpler functional language, better suited as a starting point for static analysis [4].

It is at this point in the compilation that guard expressions are wrapped in `try-catch` blocks. The Core Erlang reference specifies the desired properties of guard expressions, such as lack of side-effects and bounded —constant of linear— time of computation [5, sec. 6.7]. It also specifies that the general form of function application is not allowed in guards.

Another notion introduced by Core Erlang is that of *primitive operations*, or *primops* for short. These are similar to BIFs, in that they depend on the implementation of the underlying runtime system. Unlike BIFs however, they are not exposed as functions to the programmer.

For a more thorough understanding of Core Erlang, refer to the language reference [5]. Throughout the rest of the thesis, Erlang and Core Erlang terminology will be used interchangeably without special mention, as long as confusion does not ensue.

2.3 What is Pure in Erlang

As mentioned earlier, Erlang consists of a purely functional core, mixed with some imperative constructs. The imperative parts are mostly BIFs which deal with Input/Output operations, message passing, and other low level operations, usually related to the Erlang Virtual Machine.

2.3.1 Except exceptions

A notable gray area is exception handling in Erlang. Exceptions disrupt the normal flow of execution, as they represent a non-local return. In certain contexts however, they can be thought of as just another return value, since exceptions can be pattern-matched in special `try-catch` blocks.

Consider however the common subexpression elimination optimisation, where multiple calls to a function with the same set of arguments are replaced by a single one whose result is used in place of subsequent calls. As the example in Listing 2.5 illustrates, exceptions should be considered side-effects in such contexts.

```
1  foo(N) ->
2    42 / N.
3
4  bar(N) ->
5    X = foo(N),
6    Y = foo(N),
7    {X, Y}.
8
9  baz(N) ->
10   X = (catch foo(N)),
11   Y = foo(N),
12   {X, Y}.
13
14 bar_equiv(N) ->
15   X = foo(N),
16   {X, X}.
17
18 baz_equiv_wrong1(N) ->
19   X = (catch foo(N)),
20   {X, X}.
21
22 baz_equiv_wrong2(N) ->
23   Y = foo(N),
24   {Y, Y}.
```

Listing 2.5: Exceptions can be tricky

If we mark the function `foo/1` as pure, then we could replace the second call to it in `bar/1`, and transform `bar/1` to `bar_equiv/1`. However, in the case of `baz/1`, doing so would be wrong if exceptions are to be considered. The call `foo(0)` will generate a bad arith exception, and if we only kept `X`, this would be silenced. Conversely, keeping the version without the `catch` would also be wrong, considering a more complex example, where the two calls are in different execution paths.

One may argue that expressions like `catch` could be detected and prevent optimisation in such cases. However, handling exceptions as side-effects is undoubtedly simpler.

In contrast, exceptions do not pose a problem in case of guards, as we saw in Section 2.1.5. Furthermore, almost all Erlang functions, even total ones, are guarded with an extra catch-all clause upon translation to Core Erlang, which raises the `badarg` exception. While some of these clauses are later removed by an optimisation pass, such behaviour cannot be relied upon for two reasons: a) the optimisation pass is optional, and b) type analysis is required to accurately identify such cases. The Erlang compiler does not keep any type information, and type analysis is out of the scope of Purity. Consequently, if exceptions were considered impure, almost any function would qualify as impure too.

In any case, our choice is to leave the decision to the user of Purity, allowing exceptions to be considered as side-effects as well as pure constructs, depending on the context.

2.3.2 Examples

A non-exhaustive table of impure constructs, along with a brief explanation, is available in Table 2.1.

Function/Construct	Description
receive	Pattern match on messages, concurrency
! / erlang:send/2	Send messages, concurrency
register/2	Register a name for an Erlang process, modifies VM state
io:format/2	Print to standard output, I/O
file:read/2	Read from file, I/O
date/0	Return the current date, depends on the environment
put/2	Store value in the process dictionary, modifies state
make_ref/0	Create a unique identifier, depends on hidden state in the VM

Table 2.1: Common impure constructs in Erlang

Chapter 3

Analysis

3.1 Description

Our analysis is relatively straightforward. It operates on ERLANG modules, which are first compiled to CORE ERLANG, and consists of two distinct stages. At the information gathering stage, the CORE ERLANG *Abstract Syntax Tree* is traversed two times. The first pass collects variable aliases for each function, while the second pass associates each function analysed with a list of dependencies and a small context for each dependency. Since all impure ERLANG constructs are translated to specific function calls in CORE ERLANG —with the exception of the `receive` expression which needs special handling— this list of dependencies essentially represents a call graph.

The second stage, the core of the analysis, tries to convert each function's list of dependencies to a value representing its purity. This is achieved by means of an iterative process which ends when a fixed point is reached for all functions. The process starts by selecting the subset of functions in the call graph whose purity is predetermined, the so-called *seed*. The purity of each member of the seed is then propagated to the set of functions which depend on it, following this simple rule: if impure, the dependent function is characterised as impure too; if pure, this particular dependency is removed from the dependent function's dependency list. If a function ends up without any dependencies it can be safely marked as pure. The process is then repeated, with the seed becoming the set of functions whose purity was just determined.

The final product of our analysis is a lookup table, which maps functions to their purity. The key is a 3-tuple of *module*, *function* and *arity*, while the value can be either a concrete result, i.e., true or false, or a list of dependencies, which means the analysis was inconclusive. From a conservative standpoint such functions are considered impure as well.

To see how the analysis works in practice, some simple ERLANG functions are listed along with their results after both the first and second stage of the analysis.

```
1 % depends: [{remote,{erlang,is_list,1},[]},{remote,{erlang,is_tuple,1},[]}]
2 % is_pure: true
3 tol(L) when is_list(L) ->   true;
4 tol(T) when is_tuple(T) ->  true;
5 tol(_) -> false.
```

Listing 3.1: A genuinely pure function

Listing 3.1 presents a small function which performs a simple check on its argument. This function calls two BIFs, `is_list/1` and `is_tuple/1`, which are known to be pure. Thus, the function is pure as well. It also happens that this function is total, as it works for any type of input argument. Therefore, it cannot raise a function clause exception, so it is pure regardless of how exceptions are treated.

```
1  % depends: [{primop,{match_fail,1},[]}]
2  % is_pure: true
3  check_version(#plt{version = ?VERSION}) ->
4      ok;
5  check_version(#plt{}) ->
6      error.
7
8  % depends: [{primop,{match_fail,1},[]},{remote,{erlang,'+',2},[]}]
9  % is_pure: true
10 total(#stats{p = P, i = I, u = U, l = L}) ->
11     P + I + U + L.
12
13 % depends: [{primop,{match_fail,1},[]},{remote,{erlang,'=:=',2},[]}]
14 % is_pure: true
15 assoc_find(_Key, []) ->
16     error;
17 assoc_find(Key, [{Key, Val}|_T]) ->
18     {ok, Val};
19 assoc_find(Key, [_H|T]) ->
20     assoc_find(Key, T).
```

Listing 3.2: Pure functions which may raise an exception

Listing 3.2 presents more pure functions. Unlike the previous listing, all of these may raise exceptions, if called with inappropriate arguments. Notice how `check_version/1` expects a `#plt` record as argument, while `total/1` a `#stats` one. The compiler compensates by adding a call to the `match_fail/1` primop, which generates the appropriate exception, as a catch-all clause on pattern matches.

The addition operation, which `total/1` depends on, will also generate such an exception for non-numeric arguments, but is otherwise pure. Finally, `assoc_find/3` will raise an exception if its third argument is not a list. To provide some insight into the dependencies of `assoc_find/3`, Listing 3.3 shows the same function translated to Core Erlang. One can see how the `=:=` dependency corresponds to the pattern match on the `Key` variable of the second case clause. It also shows how the call to the `match_fail/1` primop is automatically generated by the compiler.

It follows that none of these functions will be considered pure if exceptions are to be treated as impure.

Having seen what qualifies as a pure function, it is time to take a look at what fails to do so. Listing 3.4 shows how one may construct a stateful function in Erlang. This particular example makes explicit calls that manipulate the process dictionary —which provides global mutable state for each Erlang process.

On the other hand, the functions in Listing 3.5 are all impure because they make use of concurrency primitives. First of all, the `start/0` function calls the `spawn/1` BIF, which creates a new process —clearly a side-effect— and returns its process identifier. Then, there is the call to `register/2`, which adds an appropriate entry to the process registry; another side-effect.

In similar fashion, the `loop/1` and `pcount/0` functions are impure because of their

```
'assoc_find'/2 =
    fun (_cor1,_cor0) ->
    case _cor0 of
      <[]> when 'true' ->
          'error'
      <[{_cor5,Val}|_cor6]> when call 'erlang':'=:='(_cor5,_cor1) ->
          {'ok',Val}
      <[_cor7|T]> when 'true' ->
          apply 'assoc_find'/2(_cor1, T)
      ( <_cor2> when 'true' ->
        primop 'match_fail'({'case_clause',_cor2}) -| ['compiler_generated'] )
    end
```

Listing 3.3: Part of Listing 3.2 translated to CORE ERLANG

```
% depends: [{primop,{match_fail,1},[]},{remote,{erlang,'+',2},[]},
%           {remote,{erlang,get,1},[]},{remote,{erlang,is_integer,1},[]},
%           {remote,{erlang,put,2},[]}]
% is_pure: {false,"call to impure erlang:get/1, erlang:put/2"}
count() ->
    case get(counter) of
        undefined ->
            put(counter, 1),
            1;
        N when is_integer(N) ->
            put(counter, N + 1) %% put/2 also returns the previous value.
    end.
```

Listing 3.4: A stateful accumulator

dependency to the `receive` construct, and the `!` primitive. The latter is more clear in the `stop/0` function. Other than that, we can see that `loop/1` is structured in a manner more fitting to a pure function, passing its state as argument to the next call. One should also note the `'start-1'/0` function, which represents the anonymous function passed as an argument to `spawn/1`. This is also impure, since it calls `loop/1`.

3.1.1 Bootstraping the analysis

As previously mentioned, we need an initial set of functions whose purity is predetermined. This set should include all built-in functions, since these are implemented in C, not in ERLANG, and thus cannot be analysed. It follows that we need to somehow hard-code the purity of ERLANG BIFs. To this purpose, we extracted all the BIFs in BEAM and generated an appropriate ERLANG module which holds all the necessary information. This was extended to other commonly used functions for which our analysis could not extract satisfactory results.

3.1.2 PLT: the Persistent Lookup Table

Besides having hard-coded values for functions, it is useful to keep track of previously analysed modules, allowing for incremental analysis and saving the user the trouble of re-analysing every module his application depends on, each time.

This is implemented with a persistent lookup table, or PLT for short. This PLT stores the original list of dependencies produced by the analysis of each function, along with a

```
% depends: [{remote,{erlang,register,2},[]},
%           {remote,{erlang,spawn,1},[{1,{test,'start-1',0}}]}]
% is_pure: {false,"call to impure erlang:register/2, erlang:spawn/1"}
start() ->
  register(acc, spawn(fun() -> loop(0) end)).

%'start-1'/0
% depends: [{local,{test,loop,1},[]}]
% is_pure: {false,"call to impure test:loop/1"}

% depends: [{remote,{erlang,'!',2},[]}]
% is_pure: {false,"call to impure erlang:'!'/2"}
stop() ->
  acc ! stop.

% depends: [{erl,'receive'},{remote,{erlang,'!',2},[]},
%           {remote,{erlang,self,0},[]}]
% is_pure: {false,"receive"}
pcount() ->
    acc ! {self(), next},
    receive {count, N} -> N end.

% depends: [{erl,'receive'},{remote,{erlang,'!',2},[]},
%           {remote,{erlang,'+',2},[]}]
% is_pure: {false,"receive"}
loop(N0) ->
    receive
        {Pid, next} ->
            N1 = N0 + 1,
            Pid ! {count, N1},
            loop(N1);
        stop -> ok
    end.
```

Listing 3.5: Concurrency is a side-effect

table of the final purity results, which serves as a cache for faster execution on subsequent runs. On each run, modules in the PLT are checked for modifications and re-analysed if necessary.

Our implementation is inspired by the PLT in Dialyzer, but is somewhat simpler and more limited in scope.

3.1.3 Mutually recursive functions

One problem with our dependency-driven approach relates to mutually recursive functions. Given two mutually recursive functions, with no dependencies other than the ones between them, the algorithm previously described would result in an endless loop, unable to determine the purity of either one. Worse, it is entirely possible that this can extend to a bigger set of functions, for example that `f1` depends on `f2` which depends on `f3`, which in turn depends on `f1`.

To address the issue, an extra step is added to the core of the analysis. Since mutually dependent functions form cycles in the call graph, we map each function to the particular cycle it belongs to and then single out those functions whose dependencies are all part of that same cycle. These functions can be safely marked as pure.

```
%% Directly dependent mutually recursive functions.
f(0) -> 1;
f(N) when N > 0 -> N - m(f(N-1)).

m(0) -> 0;
m(N) when N > 0 -> N - f(m(N-1)).

%% Indirectly dependent mutually recursive functions.
ping(0) -> ping_win;
ping(N) -> shoot(ping, N-1).

shoot(ping, N) -> pong(N);
shoot(pong, N) -> ping(N).

pong(0) -> pong_win;
pong(N) -> shoot(pong, N-1).
```

Listing 3.6: Examples of mutually recursive functions

3.2 Higher Order Functions and Limitations

To better understand the limitations of our analysis, we consider the case of higher order functions. A function is considered to be higher order, if it accepts other functions as arguments or returns functions as return values. Considering the first part of the definition, if a call is made to one of these arguments in the body of the higher order function, then it follows that its purity depends on it, and the purity of the function cannot be resolved to a fixed value. The only exception to this is when the function depends on other *impure* functions as well.

Let us consider a higher order function, `h`, which just makes a call to its first argument, for the sake of example. Clearly this has an unfixed purity. But what can be said about a function `g`, which depends on `h`? This function would either be a higher order function itself, taking another function as argument, and passing it along to `h`, or it would pass a *concrete* function `f`, as argument to `h`. It is thus possible in the second case —assuming the purity of `f` is known— to resolve the purity of this specific instance of `h` and consequently that of `g`. Listing 3.7 gives an example of the above.

```
%% A higher order function, which depends on its first argument.
fold(_Fun, Acc, []) ->
  Acc;
fold(Fun, Acc, [H|T]) ->
  fold(Fun, Fun(H, Acc), T).

%% A pure argument is passed to the higher order function, so the result
%% is pure as well.
g1() ->
  fold(fun erlang:'*'/2, 1, [2, 3, 7]).

%% An impure argument is passed to the higher order function, so the
%% result is impure.
g2() ->
  fold(fun erlang:put/2, computer, [ok, error]).
```

Listing 3.7: An example of a higher order function

This is a fairly simple case, but describes the most common use of higher order functions in ERLANG. Our analysis also supports multiple function arguments. The next most frequent case encountered, relates to the other possibility discussed earlier, i.e., that there are multiple levels of indirection between the initial higher order function with the dependency to its argument and the one which actually passes a concrete value. An example of this is presented in Listing 3.8.

```
%% One level of indirection: it is not apparent this is a higher
%% order function, since no call to its argument is made.
fold_1(Fun, Acc, Lst) ->
  fold(Fun, Acc, Lst).

%% Two levels of indirection, and the function argument has changed
%% position as well.
fold_2(Lst, Fun) ->
  fold_1(Fun, 1, Lst).

g3() ->
  fold_1(fun erlang:put/2, ok, [computer, error]).
g4() ->
  fold_2([2, 3, 7], fun erlang:'*'/2).
```

Listing 3.8: A more elaborate example of higher order functions

The fact that both `fold_1/3` and `fold_2/2` are higher order functions is not immediately apparent when their code is traversed. We can only infer that they depend on `fold/3`, whose purity may not be available yet. So, during the second stage of the analysis, the flow of arguments towards known higher order functions is analysed, and functions like `g3/0` and `g4/0` can be resolved to impure and pure respectively. Some real world examples of this case include functions like `ordsets:fold/3` which depends in turn on `lists:fold/3`, and `dict:fold/3` which depends on helper functions internal to the `dict` module.

The current implementation has limited support for this type of data flow analysis. It only supports indirect higher order functions which take a single argument and have no other unresolved dependencies. While the first limitation is probably straightforward to overcome, the latter can be further complicated by mutually recursive calls, a common practice for higher order functions which traverse tree-like structures for instance. An example of some mutually recursive higher order functions is presented in Listing 3.9.

Yet another limitation arises from higher order functions which receive their arguments indirectly, as parts of complex data structures, like records for instance. Again, it could be possible to detect some of these cases statically, but most of them would require runtime information. Refer to Listing 3.10 for some examples.

Another type of limitation regards higher order functions which are conditionally dependent on their arguments. For instance the example in Listing 3.11 shows how we cannot resolve the purity of `baz/1`, without a combination of control and data flow analysis. While the example may seem slightly contrived, this type of limitation is actually encountered in functions of the standard library, for instance the `io_lib:format/2` function.

In regard to higher order functions which return other functions our analysis is even more limited, as such return values are not tracked at the moment. As Listing 3.12 illustrates, while both `foo/1` and the function it returns are pure, the purity of `bar/0` cannot be determined, as it seemingly depends on an unknown value.

```
%% Mutually recursive higher order functions, adapted from the
%% 'cerl_trees' module.
treefold(Fun, Acc, Node) ->
  case node_type(Node) of
    clause ->
      listfold(Fun, Acc, clause_elements(Node));
    module ->
      pairfold(Fun, listfold(Fun, module_defs(Node),
                      listfold(Fun, Acc, module_attrs(Node)));
    cons ->
      treefold(Fun, treefold(Fun, Acc, cons_hd(Node)), cons_tl(Node));
    ...
  end.

listfold(Fun, Acc, [N|Ns]) ->
  listfold(Fun, treefold(Fun, Acc, N), Ns);
listfold(_, Acc, []) ->
  Acc.

pairfold(Fun, Acc, [{N1, N2}|Rest]) ->
  pairfold(Fun, treefold(Fun, treefold(Fun, Acc, N1), N2), Rest);
pairfold(_, Acc, []) ->
  Acc.
```

Listing 3.9: Mutually recursive higher order functions

```
%% Two function arguments, one direct and one extracted from the
%% #st record. Adapted from 'erlsom_sax_lib'.
foo(Val, Prev, ParseFun, #st{continuation = CFun, cont_state = CSt} = St) ->
  case CFun(CSt) of
    {Prev, _} ->
      throw({error, "Unexpected end of data"});
    {Next, NextContState} ->
      ParseFun(Val, Next, St#st{cont_state = NextContState})
  end.

%% Even more obscure, the funtion is extracted from an abstract data
%% type, a dictionary in this case.
bar(Key, Arg, Store) ->
  Fun = dict:fetch(Key, Store),
  case Fun(Arg) of
    {ok, Val} ->
       {Val, dict:erase(Key, Store)};
    error ->
       {error, Store}
  end.
```

Listing 3.10: Functions as parts of complex data structures

Finally, a different kind of limitation is described with the use of an example in Listing 3.13. While direct assignments of functions to variables are detected by Purity, more complex cases like that of conditional assignment are currently unsupported.

```
foo(Term, Format) when is_function(Format, 1) ->
  term_to_binary(Format(Term));
foo(Term, no_fun) ->
  term_to_binary(Term).

baz(Term) ->
  foo(Term, no_fun).
```

Listing 3.11: Conditional higher order functions

```
%% A simple closure.
foo(A) ->
  fun(B) -> A + B end.

bar() ->
  F = foo(2),
  F(40).
```

Listing 3.12: Functions as return values

```
%% We cannot handle the complex variable-binding expression generated by
%% the case clause, so F1 and F2 are not matched to either of the two
%% pure functions they may be bound to.
foo(Type, Arg) ->
  {F1, F2} =
    case Type of
      normal ->
        {fun math:sin/1, fun math:cos/1};
      reverse ->
        {fun math:cos/1, fun math:sin/1}
    end,
  {F1(Arg), F2(Arg)}.
```

Listing 3.13: Example of limitations on complex variable assignments

3.3 A Conservative Approach

It is important to note that PURITY takes a conservative approach in regard to its analysis, which implies that any function which cannot be analysed conclusively is to be considered impure. This is necessary since false positives are not acceptable, while false negatives can be tolerated. In regard to the intended use of PURITY, which is to allow for arbitrary pure functions in guards, this does not pose a significant problem. Such functions will most likely be small and straightforward.

3.4 PURITY in Numbers

ERLANG programming guidelines suggest writing relatively small and self contained pure functions, and then concentrating necessary side-effects to a few impure ones. In order to measure the effectiveness of our analysis and also gain some insight as to how closely such guidelines are actually followed, we analysed a diverse set of open source ERLANG applications:

Open Telecom Platform (OTP) An extensive collection of libraries and applications distributed with open source ERLANG. Among other things, it includes the ERLANG compiler itself, the standard library, static analysis tools like DIALYZER, an XML parsing library, and various networking tools.

CouchDB A document oriented distributed database system.

Wings3D A subdivision modeler, used for generation of polygon models in computer graphics.

ejabberd A server implementation of the Extensible Messaging and Presence Protocol (XMPP), an open standard used primarily for instant messaging.

Yaws A high performance HTTP 1.1 server.

ibrowse An HTTP 1.1 client, also a dependency of Yaws.

Erlssom Another XML parsing library and dependency of Yaws.

PURITY itself is also analysed and presented with the rest of these applications. Table 3.1 includes some further information about each application, while Table 3.2 presents the results of the analysis.

The first two columns of Table 3.2, 'Pure' and 'Impure', should be self-explanatory. Both the 'Undefined' and 'Limited' columns list functions which lack concrete pureness values. While the first represents functions whose purity is inherently undecidable, the latter contains those for which we cannot draw any conclusions because of limitations in our analysis. A typical example of the first category is a function like `apply(Module, Function, Args)`, which returns the result of calling `Function` in `Module`, with `Args`. Keeping in mind that functions in ERLANG are characterised by their arity as well as their module and name, and since `Args` is an arbitrary list, it is not always known at compile time which function will be called. [1] In general, the purity of most higher order functions is undecidable, since it depends on their arguments. Also included here are unknown functions, and any functions which depend on them. These could be BIFs and NIFs, or just functions not included in the PLT at the time of analysis. For examples of the second category, refer to Section 3.2. Finally, the last column contains CPU times for each application analysed, measured with the `erlang:statistics/1` function.

The surprisingly high percentage of undecidable functions in `Erlsom` was traced down to the extensive use of continuation passing style in its lower level parsing functions.

A graphical representation of the results in Table 3.2 can be found in Figure 3.1 through Figure 3.4.

It is worth noting, that a completely different picture is presented when exceptions are treated as impure. The percentage of pure functions plummets to the single digit values, between 5-6%. These results are omitted, since they are pretty much uniform across every analysed application.

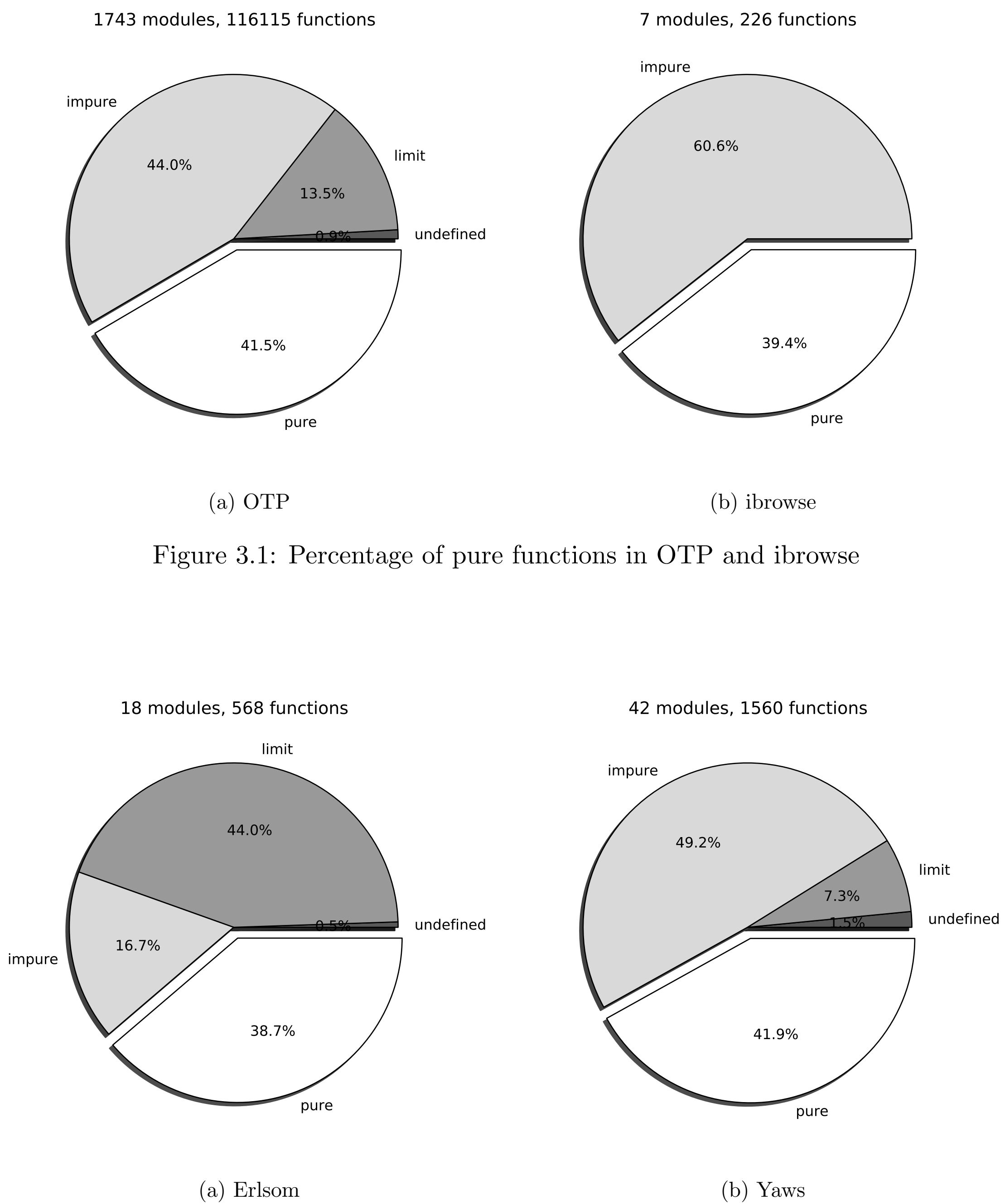
[1] In fact, calls to `erlang:apply/3` with a concrete argument list are converted to direct `Module:Function(Args)` calls by the compiler.

App	Version	Modules	Functions	LOC
OTP	R14	1,743	116,115	1,183,904
ibrowse	1.6.1	7	226	2,683
Erlsom	1.2.1	18	568	10,410
Yaws	1.88	42	1,560	30,543
ejabberd	2.1.4	149	5,168	55,457
Wings3D	1.2	168	9,506	87,774
CouchDB	0.11.0	97	2,501	22,938
purity	0.1	12	361	2,557

Table 3.1: Details of analysed applications

App	% Pure	% Impure	% Undefined	% Limited	Time
OTP	41.5	44.0	0.9	13.5	4:52
ibrowse	39.4	60.6	0.0	0.0	0:02
Erlsom	38.7	16.7	0.5	44.0	0:04
Yaws	41.9	49.2	1.5	7.3	0:07
ejabberd	35.5	56.3	4.9	3.3	0:17
Wings3D	45.8	45.3	1.2	7.8	0:27
CouchDB	42.0	46.9	1.1	10.0	0:09
purity	62.9	27.7	1.1	8.3	0:03

Table 3.2: Analysis results



(a) OTP

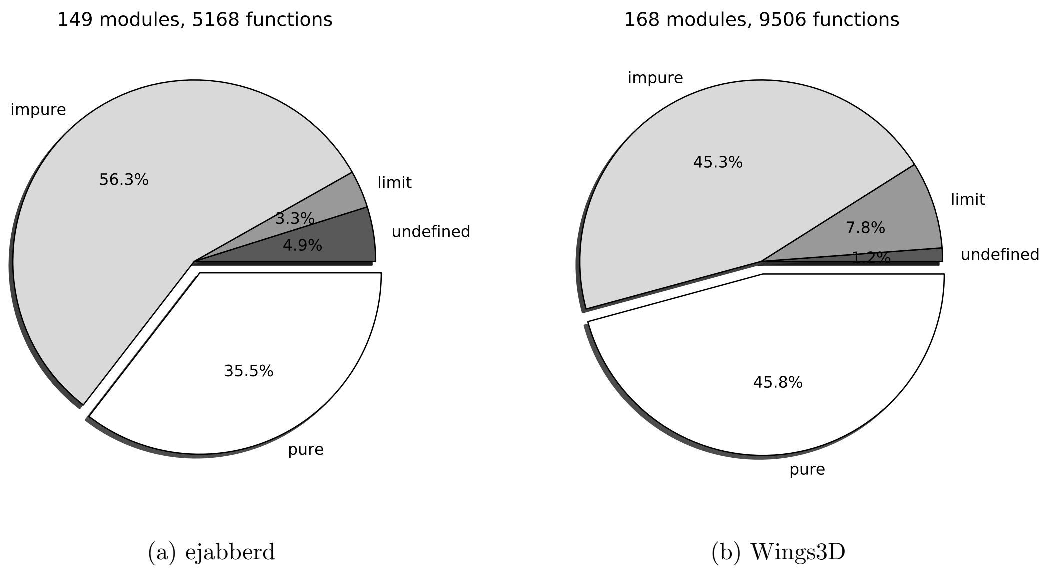
(b) ibrowse

Figure 3.1: Percentage of pure functions in OTP and ibrowse

(a) Erlsom

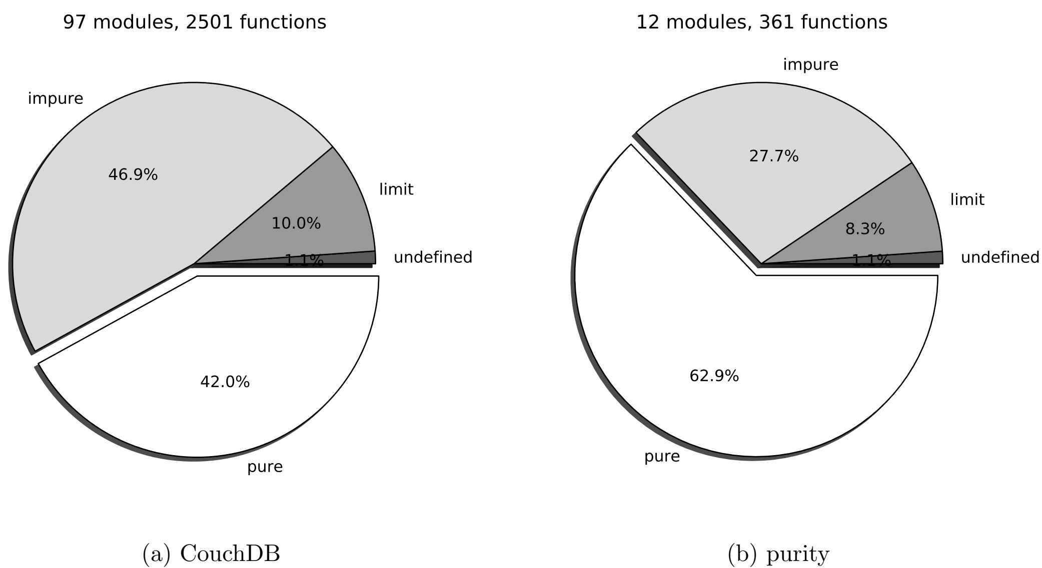
(b) Yaws

Figure 3.2: Percentage of pure functions in Erlsom and Yaws



(a) ejabberd (b) Wings3D

Figure 3.3: Percentage of pure functions in ejabberd and Wings3D



(a) CouchDB (b) purity

Figure 3.4: Percentage of pure functions in CouchDB and purity

3.4.1 Beyond the numbers

According to the previous statistics, the biggest part of most Erlang applications is written in an impure manner. However, such a conclusion is not necessarily consistent with the intentions of most Erlang programmers. To begin with, there is no measurement of the actual size of each function, and the role it plays in each module. Furthermore, if a commonly used function in a module is impure, this will propagate to every other function

calling it, and therefore most of the module.

Finally, it should be noted that our definition of purity is a strict one. Consider an example like the one in Listing 3.14. At first glance it is not so obvious whether this function is impure. Both `filename:basename/1` and `filename:rootname/1` should be simple list processing functions, and they are used for the value they return. As it turns out, they are both impure, because they depend on `os:type/0` in order to do their work in a portable manner across different operating systems. Consequently, they depend on the execution environment, and are thus not referentially transparent: their results vary across different operating systems. A typical programmer would most likely not consider the use of such a function as inconsistent with programming in a purely functional style.

This is why the possibility of distinguishing between functions which are impure because they lack referential transparency, as opposed to those which are impure because of side-effects, is being considered. This would enable the users of PURITY to select only one of the two criteria instead of both whenever that would fit better to their intended use of the results. Such an option would also help in the handling of functions like `erlang:node/0` and `erlang:node/1` which are allowed in guards —as mentioned in Section 2.1.5— but are considered impure by our analysis.

```
1 filename_to_module(Filename) ->
2     list_to_atom(filename:basename(filename:rootname(Filename))).
```

Listing 3.14: Is this function impure?

Chapter 4

User Defined Guards

4.1 Motivation

Chapter 2 introduced guards, how they are used in ERLANG, and the limitations currently enforced. Allowing arbitrary function calls as guard expressions further increases the expressiveness of the language and leaves room for more compact and descriptive code.

Grouping multiple guard tests into a single function is a readily apparent advantage. However, this *is* currently possible in ERLANG with the use of macros as shown in Listing 4.1. The real advantage of such an extension is *better abstraction*.

```
-define(is_mfa(M, F, A), is_atom(M), is_atom(F), A >= 0, A =< 255).

...
is_pure(erlang,is_integer,1) ->
    true;
is_pure(M, F, A) when ?is_mfa(M, F, A) ->
  unknown.
```

Listing 4.1: Compact guards with the help of macros

Abstract data types —often abbreviated to ADTs— are a fundamental aspect of most programming languages. They make it possible to hide the implementation details of a particular data type, and only allow it to be manipulated via well defined interfaces. For example, a set module should only export common set manipulation operations, like element addition, unions and intersections. Whether the set is implemented as a balanced tree, a hash table or something else, does not affect those fundamental operations. Keeping the data type abstract, allows the programmer to change it —to a more efficient one for instance— without breaking all the code which depends on set operations.

Unfortunately, structural information of data types is exposed in ERLANG, as it allows inspection through pattern matching and type tests. Recent additions to DIALYZER, a static analysis tool for ERLANG, allow specifying terms as *opaque*, and detecting violations of their opaqueness outside their module [14]. Such violations should be replaced with proper type tests exported by that module. For example, all three modules implementing sets in the ERLANG standard library —`sets`, `ordsets` and `gb_sets`— export an `is_set/1` function, to this end.

Allowing arbitrary type tests as guards can help make code cleaner, and could even discourage programmers from breaking ADT contracts. The two examples in Listing 4.2, depict an attempt at distinguishing between different set implementations. The first ver-

sion shows how one might implement it in ERLANG currently, while the second shows how simpler it could become with the help of user defined guards.

```
%% Custom tests not allowed as guards.
foo(Set) ->
  case gb_sets:is_set(Set) of
    true ->
      handle_gb_set(Set);
    false ->
      case sets:is_set(Set) of
        true ->
          handle_set(Set);
        false ->
          error
      end
  end.

%% Custom tests allowed.
foo(Set) when gb_sets:is_set(Set) ->
  handle_gb_set(Set);
foo(Set) when sets:is_set(Set) ->
  handle_set(Set);
foo(_) ->
  error.
```

Listing 4.2: Employing custom abstract type tests in guards

Besides type tests, functions such as `gb_sets:is_empty/1` are also prominent candidates for use as guards. Listing 4.3 presents additional examples.

```
foo(S) when gb_sets:is_empty(S) ->
  error;
foo(S) ->
  {ok, gb_sets:largest(S)}.


%% Example of how a balanced tree manipulation function could look like.
bar({tree, _, L, R} = Tree, ....) when is_balanced(Tree) ->
  handle_balanced...
bar({tree, _, L, R} = Tree, ....) when is_left_chain(L) ->
  handle_left_chain...
bar({tree, _, L, R} = Tree, ....) when is_right_chain(R) ->
  handle_right_chain..

%% Another example of type abstraction, taken from Purity.
is_impure({false, _}) ->
  true;
is_impure(Ctx) when is_context(Ctx) ->
  ctx_mem({erl, 'receive'}, Ctx);
is_impure(_) ->
  false.

is_context(C) -> is_list(C).
```

Listing 4.3: More cases of useful user defined guards

4.2 Erlang Compilation

In the course of its compilation, an Erlang program has to go through many transformations, represented by distinct passes in the compiler. These include conversions between several intermediate languages, three of which are of interest to us:

1. Core Erlang: An intermediate representation of Erlang, more suitable for analysis and code transformations;
2. Kernel Erlang: An internal representation of the compiler;
3. Kernel Erlang annotated with variable life-time information; and
4. BEAM bytecode, where our interaction with the compiler stops

4.3 Implementation

A proof of concept implementation of user defined guards was implemented as a patch against the development version of Erlang R14. It adds an optional flag to the Erlang compiler, which allows for arbitrary functions to be used as guards, but only if they are free of side-effects.

Two distinct aspects of the compilation process were altered. First of all, the compiler frontend was modified. An extra pass was added for pureness analysis, as well as command line flags to control it. Other than that, part of the compiler backend had to be changed so that arbitrary Erlang functions could be executed properly in guard expressions. A more thorough explanation of the changes follows.

4.3.1 Frontend

Errors relating to invalid guards expressions are first converted to warnings. If the `+pure_guards` flag is provided, and after the code is translated to Core Erlang, it is analysed for purity. Afterwards, any functions used in guards are looked up. If they are pure, the warning is preserved, otherwise it is reverted to an error. This is also what happens if the flag is not present.

4.3.2 Backend

Minor changes were necessary up to the code generation stage. Support for handling arbitrary call expressions in Kernel Erlang was added.

In the subsequent pass, where Kernel Erlang is annotated with lifetime information, any functions with user defined guards were marked and wrapped in a block expression. This was in order to take advantage of the existing mechanisms in the BEAM code generation phase, which insert appropriate stack and register manipulation instructions, such as allocation and deallocation.

Small modifications to the BEAM validation pass were also necessary.

4.4 Engineering Issues

A proposal for somehow annotating pure functions is being considered. This could be used to trigger appropriate compiler checks. Furthermore, it would make the programmer's intentions explicit, and help in making the use of user defined guards less confusing.

Another thing to consider is the ERLANG loader. As mentioned in Section 2.1.7, because of on-the-fly code reloading the ERLANG loader needs to make sure that a newer version of an ERLANG module does not violate the assumptions made for the earlier version. Purity annotations could assist the loader in checking for potential violations.

4.5 Usage Examples

Some examples of using the modified compiler are presented in Listing 4.4 through Listing 4.8.

```
-module(test1).
-compile(export_all).

mylist(L) when is_list(L) ->
    true;
mylist(_) ->
    false.

f(H) when mylist(H) ->
    [ok|H];
f(_) ->
    error.

h(H) ->
    if mylist(H) -> [ok|H];
       true -> error
    end.

g(H) when mylist(H) ->
    [ok|H];
g(H) ->
    H.
```

Listing 4.4: A very basic user defined guard

```
$ ./bin/erlc test1.erl
test1.erl:9: illegal guard expression
test1.erl:15: illegal guard expression
test1.erl:19: illegal guard expression

$ ./bin/erlc +pure_guards test1.erl
test1.erl:0: Warning: error loading plt: no_such_file
test1.erl:9: Warning: user defined guard mylist/1
test1.erl:15: Warning: user defined guard mylist/1
test1.erl:19: Warning: user defined guard mylist/1

$ ./bin/erlc +pure_guards '+{plt,"otp.plt"}' test1.erl
test1.erl:9: Warning: user defined guard mylist/1
test1.erl:15: Warning: user defined guard mylist/1
test1.erl:19: Warning: user defined guard mylist/1
```

Listing 4.5: Example usage of the modified compiler

As we can see in the example, a simple test module is compiled. First we pass no flags to the compiler and get an illegal guard expression error as a result. Then, we make use

of the `+pure_guards` flag, in which case we are warned of the fact that we are using user defined guards, but the program compiles fine. There is one more warning, regarding the PLT which is not found at its default location. So, in the subsequent call we pass the location of the PLT file explicitly.

While this example was simple enough not to require a PLT, a more advanced example shows how it can be useful. As illustrated in Listing 4.6 and Listing 4.7, there is no way to determine the purity of the `ordsets:is_element/2` function, since it belongs to a different module. If however the results of a previous analysis are stored in a PLT, the function can be used as a guard. The example also illustrates how the compiler will report an error if a function is not found to be pure.

```
1  -module(test4).
2  -compile(export_all).
3
4  f(L) when ordsets:is_element(x, L) ->
5      success;
6  f(_) ->
7      failure.
```

Listing 4.6: A more useful guard

```
1  $ ./bin/erlc +pure_guards test4.erl
2  test4.erl:4: illegal guard expression
3  test4.erl:0: Warning: error loading plt: no_such_file
4
5  $ ./bin/erlc +pure_guards '+{plt,"otp.plt"}' test4.erl
6  test4.erl:4: Warning: user defined guard ordsets:is_element/2
```

Listing 4.7: Different results depending on the presence of the PLT

Finally, the correctness of the `test1` module is verified by calling its functions from the ERLANG shell. This is shown in Listing 4.8.

```
1  $ ./bin/erl
2  Eshell V5.7.5  (abort with ^G)
3  1> test1:mylist([]).
4  true
5  2> test1:f([1]).
6  [ok,1]
7  3> test1:f(3).
8  error
9  4> test1:h([2]).
10 [ok,2]
11 5> test1:h({}).
12 error
13 6> test1:g({tup,le}).
14 {tup,le}
15 7> test1:g([{tup,le}]).
16 [ok,{tup,le}]
17 8>
```

Listing 4.8: Verifying the guards work as expected

4.6 Limitations

Some limitations of our prototype should be noted. To begin with, it is not possible to make use of the patch when defining functions in the ERLANG shell at the moment. Another limitation relates to calls to higher order functions in guard expressions and is described with the help of an example in Listing 4.9. While the guards in function `foo/1` and `bar/1` are equivalent, the first one will generate an *illegal guard expression* error, since the purity of `lists:all/1` is actually not fixed but depends on its argument. However, no call site analysis will be performed at this stage, the compiler merely looks up the value of the function which is part of the guard expression in the table returned by a previous analysis pass. We can easily work around this by defining a wrapper function like `literals/1`, as shown in the example. Whether the above presents an actual limitation or instead helps keep guard expressions simple and easier to comprehend is debatable.

```
1  %% Illegal guard expression
2  foo(Arg) ->
3      Args = cerl:call_args(Arg),
4      case call_mfa(Arg) of
5          {erlang,make_fun,3} when lists:all(fun cerl:is_literal/1, Args) ->
6              list_to_tuple([cerl:concrete(L) || L <- Args]);
7          ...
8      end.
9
10 %% Same functionality, but legal guard expression
11 bar(Arg)
12     Args = cerl:call_args(Arg),
13     case call_mfa(Arg) of
14         {erlang,make_fun,3} when literals(Args) ->
15             list_to_tuple([cerl:concrete(L) || L <- Args]);
16         ...
17     end.
18
19 literals(Args) ->
20     lists:all(fun cerl:is_literal/1, Args).
```

Listing 4.9: Limitations on direct calls to higher order functions

Chapter 5

Related Work

5.1 Purity in other Languages

Even though we are not aware of any related work regarding ERLANG specifically, purity concerns most functional programming languages, one way or the other.

5.1.1 The case of HASKELL

HASKELL is a purely functional, statically typed programming language, with non-strict semantics. It employs one of the most interesting and exotic approaches to functional programming, managing to keep the language pure, while allowing for arbitrary side-effects by encapsulating them into monads [12].

This approach cleanly isolates pure from impure code, and at the same time, relies on HASKELL's rich type system for determining the purity of functions. This way, no extra annotations are required, and in combination with type inference, the programmer is relieved of the burden of specifying such properties. Listing 5.1 shows how some common impure functions can be encapsulated in the IO monad and how such information is encoded in their type signature. Also noteworthy is the use of the `sequence` function, which forces the evaluation of the particular sequence of actions from left to right in order to produce the desired side-effects.

```
1  module Main where
2  import Random
3
4  say_hello :: IO ()
5  say_hello =
6    putStrLn "Hello, world!"
7
8  rand :: Int -> IO Int
9  rand n =
10   randomRIO (1, n)
11
12 main :: IO [()]
13 main = do
14   n <- rand 42
15   sequence $ take n $ repeat say_hello
```

Listing 5.1: Encapsulation of impure functions into monads

5.1.2 Clean

Clean is similar to HASKELL in many respects. It is purely functional, statically typed and also features non-strict evaluation. It differs however in its approach in regard to purity. Clean uses a uniqueness typing system [1, ch. 9]. This extends a traditional type system, by allowing the user to specify that a given function argument is unique. This annotation guarantees the function will have private access to the argument, therefore destructive updates to it will not violate the function's semantics during the execution of the program.

5.1.3 BITC

BITC was developed as a systems programming language with the goal of supporting formal verification. Unlike the previous two languages, BITC is not purely functional. It does however support user level type annotations regarding the purity of functions, through a so called effect type system [16, ch. 10]. This associates expressions with an effect variable, which can have a value of *pure*, *impure* or *unfixed*.

The use of the effect type system, and the classification of expressions is similar in many regards to the way our analysis works. BITC takes this one step further and allows for annotations meant to ensure stateless execution, leaving room for compiler optimisations.

5.1.4 Joe-E

Joe-E is a subset of the Java programming language specifically developed to support verification of pure functions via static analysis, something previously not possible in a high-level imperative language such as Java. Joe-E is described as an *object-capability* language by its designers and focuses primarily on using pureness analysis to ensure security properties such as function invertibility, reproducibility of computation and execution of untrusted code [10].

5.2 Other Uses

5.2.1 Compiler optimizations

A few compiler optimisations depend on pureness analysis. Some of the more common ones are listed here.

5.2.1.1 Common Subexpression Elimination (CSE)

As mentioned in previous sections, common subexpression elimination takes advantage of pure functions, so that multiple calculations of the same value can be avoided.

5.2.1.2 Code Reordering

This is a general type of optimisation, with many variations. The idea is to change the order of certain expressions in a program while preserving its semantics. In most cases the goal is to achieve better locality of reference, in order to exploit instruction pre-fetching and CPU caches.

5.2.1.3 Parallelization

As long as two or more pure functions do not depend on the result of each other they can be executed in parallel. With this observation in mind, it is possible for a compiler to perform a parallelizing code transformation to gain performance benefits, something all the more important in today's multi-core environments. This optimisation can be combined with code reordering for better results.

5.2.1.4 Memoization

Memoization is a type of caching optimisation, where results of functions calls are stored, and instead of evaluating the same function repeatedly, the cached results are returned. The concept is somewhat similar to the CSE optimisation described earlier, but is more general, as the stored calls may span across completely different parts of the executing program.

5.2.2 Miscellaneous

As mentioned in the introduction, one of the drawbacks of pure functions is reduced performance. While there have been advancements in the development of efficient purely functional algorithms and data structures [13], there is ongoing research towards different directions as well.

Chapter 6

Conclusion

6.1 Concluding Remarks

This thesis analysed the required properties for an Erlang function to be considered pure, in the functional programming sense, and presented Purity, a tool for determining such properties by way of static analysis of Erlang source code. As a direct application of this analysis the possibility of enhancing the Erlang language with user defined guard expressions was considered and a small —and incomplete— prototype of this was implemented as a patch against the Erlang compiler. To seriously consider such an extension however, further assurances for Erlang functions in guard expressions are necessary. In particular, evaluation in bounded time is a mandatory requirement, a property that could be determined by some type of conservative finiteness analysis. Some engineering issues need to be worked out as well.

In the course of testing our implementation diverse code bases were analysed, providing insight as to the current practices of Erlang programmers. The percentage of functions that are classified as pure by our analysis constitutes roughly 40 to 50% of the functions in all the modules of an application. This percentage is significant if one takes into account that Erlang is primarily a concurrent language; the fact that it is also a functional language is itself a side-effect.

6.2 Future Work

It is our intention to release Purity as open source software so as to to further pursue the enhancement of the Erlang language with arbitrary guard expressions. Besides support for distinguishing between functions which are impure because of lack of referential transparency as opposed to the presence of side-effects, Purity could also benefit from more advanced data and control flow analysis in order to increase its accuracy.

6.3 Acknowledgements

The source code of Dialyzer [15] has been very helpful in the development of Purity and earlier versions of the code depended on some of its modules. Various free and open source software applications, including the Erlang distribution itself, provided significant help in testing and diagnosing the functionality of Purity. Without such wealth of applications freely available in source code form, the development of Purity would have been significantly harder.

Bibliography

[1] *Clean Language Report*, November 2002. Version 2.1, http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf.

[2] Joe Armstrong. Concurrency oriented programming in erlang, February 2003.

[3] Thanassis Avgerinos and Konstantinos F. Sagonas. Cleaning up erlang code is a dirty job but somebody's gotta do it. In Clara Benac Earle and Simon J. Thompson, editors, *Erlang Workshop*, pages 1–10. ACM, 2009.

[4] Richard Carlsson. An introduction to core erlang. http://www.it.uu.se/research/group/hipe/cerl/doc/cerl_intro.ps, September 2001.

[5] Richard Carlsson, Björn Gustavsson, Eric Johansson, Thomas Lindgren, Sven olof Nyström, Mikael Pettersson, and Robert Virding. Core erlang language specification. http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf, November 2004.

[6] Maria Christakis and Konstantinos F. Sagonas. Static detection of race conditions in erlang. In Manuel Carro and Ricardo Peña, editors, *PADL*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2010.

[7] Ericson AB. *Erlang Reference Manual User's Guide*, June 2010. Version 5.8, http://www.erlang.org/doc/reference_manual/users_guide.html.

[8] Ericson AB. *Erlang Run-Time System Application (ERTS) Reference Manual*, June 2010. Version 5.8, http://erlang.org/doc/apps/erts/index.html.

[9] Ericson AB. *Interoperability Tutorial User's Guide*, June 2010. Version 5.8, http://www.erlang.org/doc/tutorial/users_guide.html.

[10] Matthew Finifter, Adrian Mettler, and Naveen Sastry David Wagner. Verifiable Functional Purity in Java. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pages 161–174, New York, NY, 2008. ACM.

[11] Wolfram Kahl. http://www.cas.mcmaster.ca/~kahl/reftrans.html.

[12] John Launchbury and Simon L. Peyton Jones. State in haskell. In *Lisp and Symbolic Computation*, pages 293–341, December 1995.

[13] Chris Okasaki. *Purely Functional Data Structures*. 1998.

[14] Konstantinos F. Sagonas. Opaque data types in erlang. Presentation at Erlang User Conference, November 2009.

[15] Konstantinos F. Sagonas. Using static analysis to detect type errors and concurrency defects in erlang programs. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS*, volume 6009 of *Lecture Notes in Computer Science*, pages 13–18. Springer, 2010.

[16] Jonathan Shapiro, Swaroop Sridhar, and M. Scott Doerrie. The origins of the bitc programming language. `http://www.bitc-lang.org/docs/bitc/bitc-origins.html`, April 2008.