View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CiteSeerX

Purity in Erlang

Mihalis Pitidis[1] and Konstantinos Sagonas[1,2]

[1] School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
mpitid@gmail.com, kostis@cs.ntua.gr

Abstract. Motivated by a concrete goal, namely to extend Erlang with the ability to employ user-defined guards, we developed a parameterized static analysis tool called PURITY, that classifies functions as referentially transparent (i.e., side-effect free with no dependency on the execution environment and never raising an exception), side-effect free with no dependencies but possibly raising exceptions, or side-effect free but with possible dependencies and possibly raising exceptions. We have applied PURITY on a large corpus of Erlang code bases and report experimental results showing the percentage of functions that the analysis definitely classifies in each category. Moreover, we discuss how our analysis has been incorporated on a development branch of the Erlang/OTP compiler in order to allow extending the language with user-defined guards.

1 Introduction

Purity plays an important role in functional programming languages as it is a cornerstone of *referential transparency*, namely that the same language expression produces the same value when evaluated twice. Referential transparency helps in writing easy to test, robust and comprehensible code, makes equational reasoning possible, and aids program analysis and optimisation. In pure functional languages like Clean or Haskell, any side-effect or dependency on the state is captured by the type system and is reflected in the types of functions. In a language like ERLANG, which has been developed primarily with concurrency in mind, pure functions are not the norm and impure functions can freely be used interchangeably with pure ones. Still, even in these languages, being able to reason about the purity of functions can prove useful in various situations.

This paper discusses properties that functions must satisfy in order to be classified as having a certain level of purity, and describes the design and implementation of a fully automatic parameterized static analyzer, called PURITY, that determines the purity of ERLANG functions. Although the analysis is simple and very conservative, we were able to determine the purity of roughly 90% of the functions in the code bases we tested.

As a practical application, our analysis has been integrated in a development branch of the ERLANG compiler, allowing functions that the analyzer determines as pure to be used in guard expressions, something not previously possible in ERLANG and, for many years now, one of the most frequent user requests for extending the language. Furthermore, our analysis could make way for some types of optimisations in the ERLANG compiler including common subexpression elimination, useless call elimination, deforestation and automatic parallelization.

J. Hage, M.T. Morazán (Eds.): IFL 2010, LNCS 6647, pp. 137–152, 2011.
© Springer-Verlag Berlin Heidelberg 2011

The contributions of this paper are as follows:

- we present a relatively simple but parameterized static analysis that can determine the purity of ERLANG functions at different levels;
- we give detailed measurements of percentages of functions that our analysis classifies as definitely pure; to the best of our knowledge this is the first time that such numbers are reported in the literature (especially for a dynamically typed language); and
- we discuss how the analysis has served as a basis for allowing the ERLANG language to be extended with the ability to employ user-defined guards.

The next section reviews the ERLANG language and aspects of its evolution and implementation which are relevant to the topic of the paper. Section 3 describes the analyses we employ to determine the purity of ERLANG functions, followed by a Section 4 which presents experiences from running these analyses on a large corpus of ERLANG code bases. Section 5 describes how the analysis information can be used to allow for user-defined guards in ERLANG and the paper ends with reviewing purity in other languages (Section 6) and some concluding remarks.

2 Erlang: The Language and Its Features

ERLANG is a concurrent functional programming language with dynamic types. What sets ERLANG apart from other functional languages is its support for concurrency, fault tolerance and distributed programming. Other notable features include hot-code reloading whereby the code of some module of an executing ERLANG program can be replaced with a newer version of that module without interrupting the program's execution. The language also provides soft real-time guarantees.

The aforementioned features make ERLANG ideal for building highly scalable, reliable and robust systems. While initially conceived to develop software for telecommunication systems, ERLANG has outgrown this particular niche and with the advent of the multi-core era it is being used for the development of a growing number of diverse software applications. This includes web and chat servers, distributed document stores, and network servers.

ERLANG employs a mixture of purely functional programming, in the form of immutable data structures and single assignment variables, combined with a limited set of impure functions and expressions, in order to support concurrency and distribution. In particular, ERLANG implements the *actor* model of concurrency [1]. Its implementation can be summarised as concurrency based on lightweight processes communicating via asynchronous message passing with copying semantics. This helps express complex concurrency schemes in a more natural and declarative manner.

Impurities in ERLANG originate from particular expressions and functions. An example of the former is the `receive` expression which is used to extract messages from the mailbox of a process. For examples of the latter, we first need to mention the general concept of built-in functions, or BIFs as they are usually known in the ERLANG community. BIFs are functions native to the ERLANG virtual machine, implemented in the language the VM is written in, in this case C. Besides some primitive operations

which otherwise cannot be expressed in pure ERLANG, BIFs often substitute commonly used functions for optimisation purposes. As it happens, many BIFs are impure, usually because they interface with the runtime system in various ways.

Like many functional languages, ERLANG supports pattern matching, a way of matching a sequence of values against a corresponding sequence of patterns. The result, if successful, is a mapping of variables from the first pattern that matches to the various terms in the sequence of values. Pattern matching plays a central role in expressing control flow in ERLANG. Additional constraints can be placed on pattern matches with the use of guard tests. Guard tests consist of boolean expressions which are evaluated for each pattern matched and only if their result is true will the match be successful. With guards it is possible to extend the expressiveness of pattern matching significantly, e.g. to add support for value ranges for numbers, or tests for abstract values like process identifiers and function objects.

However, ERLANG currently imposes strict limitations on guard tests. Specifically, they must lack side-effects and execute in bounded time, preferably constant. To this end, guards are limited by the ERLANG language to a small predefined set of built-in functions also known as guard BIFs [2, § 6.20, p. 103].

The example in Listing 1 showcases the use of pattern matching in ERLANG and how it is further extended with guard tests. This combination allows for concise and declarative code, offering a significant boost in programmer productivity.

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

Listing 1. Examples of pattern matching and guard tests

As mentioned, ERLANG is dynamically typed. Furthermore, the ERLANG compiler currently does not perform any form of type analysis. This has certain implications on PURITY, which are discussed in detail in Section 3.

3 Purity Analysis in Erlang

In order to determine the purity of ERLANG functions we designed and implemented a fully automated static analysis which operates on ERLANG source code (or compiled bytecode files which include debugging information). The analysis is flexible and allows the user to select between different purity criteria, depending on the intended use of the analysis' results.

3.1 Flavours of Purity

We should first clarify what we mean by pure and impure functions. A *pure* function is one that is *referentially transparent*, i.e., calls to the function can be replaced by their

return values without changing the semantics of the program in any way in any execution environment. This is the strongest definition of purity that our analysis supports. In addition, it offers a choice between a few progressively weaker criteria, depending on the intended use of the analysis' results.

In general, a function may lose its referential transparency and be classified as impure: a) either due to *modifying* the execution environment in some way other than returning a value, or b) by *depending* on the environment of execution in some way other than its arguments. A function that falls into the first category is said to have side-effects. Such a function will always be considered impure by our analysis. Regarding the second category, certain uses of the analysis may choose to ignore such violations of referential transparency and force the analysis to consider functions which fall in it as pure. We shall elaborate on this design decision in a later section.

Besides the fundamental categories described above, our analysis distinguishes between yet another condition of purity, one that is more specific to ERLANG. This condition concerns *exceptions*, which represent a non-local return out of a function and are somewhat problematic in their classification. First of all, it is not clear whether exceptions break referential transparency. While we can no longer replace the function by its value, we can still replace it by an exception raising expression, preserving the semantics of the program. This is not the whole truth however, since exceptions usually carry context sensitive information, specifically the series of function calls leading up to them, otherwise referred to as a *stack trace*. In the case of ERLANG, exceptions are regular terms that can be pattern matched on. The stack trace may or may not be part of the exception value, depending on the expression used to *catch* it. The older `catch` expression converts exceptions to tuples which often contain this stack trace, so using `catch` will break referential transparency. The newer and more robust `try-catch` construct [4] however, does not directly capture the stack trace, which is otherwise available through a specific ERLANG built-in function, aptly named `get_stacktrace()`. So, in the absence of a call to this function, `try-catch` blocks can be considered pure.

Still, when it comes to certain uses of the analysis' results — such as optimisations like common subexpression elimination — it makes sense to consider all exceptions as impure. This is why our analysis is flexible and parameterized in this respect as well.

A final note regarding exceptions concerns the semantics of process termination in ERLANG. If a process is terminated by an exception which is not a member of the `exit` class, then this event is reported by the ERLANG runtime system to the error logging service [4, § 2.7]. We choose to ignore this potential side-effect since it does not directly influence the execution of the program, but primarily because we wish to maintain a conceptual separation between exceptions and side-effects. This is important, as we will see in Section 5 that exceptions can be safely ignored in certain contexts.

To sum up, the analysis we will describe in the rest of this section will classify functions as impure based on three progressively stronger criteria of impurity, namely considering a function as impure if it:

1. contains side-effects — this is the default;
2. contains no side-effects but has dependencies on the environment; and
3. contains no side-effects, has no dependencies on the environment, but possibly raises exceptions.

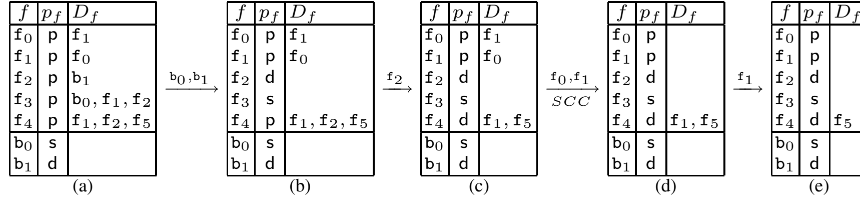
3.2 The Core of the Analysis

Our analysis is relatively straightforward. It operates on a set of ERLANG modules which are first compiled to CORE ERLANG [3] (an intermediate, simpler representation of ERLANG source code). The analysis consists of two distinct stages: an information gathering and an information propagation stage.

The first stage collects necessary information by traversing the *Abstract Syntax Tree* of each function and constructs its *dependency set*: the set of other functions it calls somewhere in its body. In our implementation, functions are identified with triples of the form $\{\mathtt{m}, \mathtt{f}, \mathtt{a}\}$ (where `m` is a module name, `f` is a function name, and `a` is its arity). However, for convenience in this section we will use shorter identifiers for functions: $\mathtt{f}_0, \mathtt{f}_1, \ldots$ for those that are analyzed and $\mathtt{b}_0, \mathtt{b}_1, \ldots$ for those that are BIFs or previously analyzed and their purity level is *a priori* known to the analysis.

All analysis information is kept in a *lookup table* which is a hash table whose keys are the function identifiers f and contains as values the *purity level* p_f of each f and D_f the *dependency set* of f. Purity levels are elements from the domain $\{\mathtt{s}, \mathtt{d}, \mathtt{e}, \mathtt{p}\}$ representing functions which contain side-effects (`s`), are side-effect free but have a dependency on the environment (`d`), are side-effect free and independent from the environment but possibly raise exceptions (`e`), or are pure (`p`). The analysis domain is thus ordered as: $\mathtt{s} > \mathtt{d} > \mathtt{e} > \mathtt{p}$. We will denote by $sup(p_{\mathtt{f}_1}, p_{\mathtt{f}_2})$ the supremum of purity levels $p_{\mathtt{f}_1}$ and $p_{\mathtt{f}_2}$ and by $sup(F)$ the supremum of purity levels of a set of functions F. Note that the purity level of a function is fully determined only when its dependency set is empty; for a function with a non empty dependency set D_f, its purity level is still conditional on the functions which appear in D_f. As we will see, during the information propagation stage of the analysis, dependency sets will be decreasing while purity levels will be increasing. If a function's purity level ever reaches the maximal value (`s`), its dependency set is not needed anymore and is removed by the algorithm.

Let us see the analysis on an example; cf. Figure 1(a). In the first stage, the analysis has scanned the code of five functions $\mathtt{f}_0 \ldots \mathtt{f}_4$ and has constructed their dependency sets. In our example, function $\mathtt{f}_2$ depends on the built-in function $\mathtt{b}_1$, while $\mathtt{f}_4$ depends on functions $\mathtt{f}_1$ and $\mathtt{f}_2$ which will be analyzed and on function $\mathtt{f}_5$ which was not given to the analysis. The purity level of all built-in functions is *a priori* known to the analysis; see the bottom part of the tables in Fig. 1. The purity level of all functions that will be analyzed is initialized to `p`. The analysis maintains as its working set the functions whose purity level is fully determined (i.e. those with empty dependency sets). For each of them it propagates its purity level to functions that depend on it, "contaminating" them with their (im)purity level in the process. After having done so, it also removes the function from the dependency set or completely removes the dependency set if the function it contaminated has reached the highest level of impurity (`s`). In our example, functions with known purity level are the two BIFs and by propagating their information we end up with the table of Fig. 1(b). Note that $D_{\mathtt{f}_3}$ has been set to empty because it has reached the highest level of impurity. Functions $\mathtt{f}_2$ and $\mathtt{f}_3$ join the working set and their use by the information propagation stage results in the table of Fig. 1(c). At this point the working set of the analysis is empty, but there are still some functions whose purity level is still conditional on functions which are part of the analysis. To achieve further progress the analysis finds an independent strongly connected component (SCC), i.e.,



(a)

f	p_f	D_f
f_0	p	f_1
f_1	p	f_0
f_2	p	b_1
f_3	p	b_0, f_1, f_2
f_4	p	f_1, f_2, f_5
b_0	s	
b_1	d	

$\xrightarrow{b_0, b_1}$

(b)

f	p_f	D_f
f_0	p	f_1
f_1	p	f_0
f_2	d	
f_3	s	
f_4	p	f_1, f_2, f_5
b_0	s	
b_1	d	

$\xrightarrow{f_2}$

(c)

f	p_f	D_f
f_0	p	f_1
f_1	p	f_0
f_2	d	
f_3	s	
f_4	d	f_1, f_5
b_0	s	
b_1	d	

$\xrightarrow[SCC]{f_0, f_1}$

(d)

f	p_f	D_f
f_0	p	
f_1	p	
f_2	d	
f_3	s	
f_4	d	f_1, f_5
b_0	s	
b_1	d	

$\xrightarrow{f_1}$

(e)

f	p_f	D_f
f_0	p	
f_1	p	
f_2	d	
f_3	s	
f_4	d	f_5
b_0	s	
b_1	d	

Fig. 1. Illustration of the analysis algorithm on an abstract example

a set of functions mutually dependent on each other, but on no other functions outside the SCC, sets their purity level to the supremum of their purity levels, and simplifies the dependency sets of these functions. In our example, functions f_0 and f_1 form such an SCC. (If functions in the SCC had different purity levels, which is not the case here, they would all collapse to their supremum at this point.) Simplifying their dependency sets results in the table of Fig. 1(d) and adds f_0 and f_1 to the analysis' working set. After a further simplification using f_1 (f_0 has no effect here), the final result of the analysis on our example is shown in Fig. 1(e).

We can now present the information propagation stage of the analysis in more detail. Algorithm 1 shows its pseudocode which is self-explanatory. What is not shown in that pseudocode is that, for efficiency reasons, the actual implementation maintains a *working set*. Whenever a function's dependency set becomes empty, the function is added to this working set. Another data structure maintained by the analysis is the *active function dependency graph*: its nodes are functions with non-empty dependency sets and its edges are formed by the elements in these dependency sets. This graph is used to find independent SCCs whenever needed.

Algorithm 1. The information propagation stage of the analysis

repeat
  for each function e in the lookup table with $D_e = \emptyset$
    for each function f where $e \in D_f$
      p_f := sup(p_f, p_e)
      if p_f = s **then** D_f := $\emptyset$ **else** D_f := $D_f \setminus \{e\}$
  F := an independent strongly connected component
  for each f in F
    p_f := sup(F)
    D_f := $D_f \setminus F$
until there are no more changes to the lookup table

The result of the analysis is left in the lookup table where the purity level of some functions may still be conditional; i.e. their dependency sets may still contain some elements, like the dependency list of function f_4 in Fig. 1(e). Note that in this way the analysis is also able to handle incomplete programs. Subsequent uses of the analysis' results must typically consider this information in a conservative way: all functions with unknown dependencies are viewed as belonging to the highest level of impurity.

3.3 Higher Order Functions

Higher order functions, i.e. functions that return other functions or accept functions as arguments, are common in functional programming languages. Considering the latter type of higher order functions, if a call is made to one of the arguments in the body of the higher order function, it follows that its purity depends on that argument, and 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 the example of a higher order function, h, which just makes a call to its first argument. 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 2 shows such an example; the comments in the code provide the necessary explanations.

```
%% 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 closure is passed to a higher order function
%% so function g1/0 will be determined pure by the analysis.
g1() -> fold(fun erlang:'*'/2, 1, [2, 3, 7]).

%% An impure closure is passed to a higher order function
%% so function g2/0 is classified as impure.
g2() -> fold(fun erlang:put/2, computer, [ok, error]).
```

Listing 2. An example showing the treatment of higher order functions

This is a fairly simple example, but it manages to capture the most common use of higher order functions in ERLANG. To handle higher order functions the implementation extends the dependency sets to also contain information about argument positions that may contain function closures that will be called. Another important case, albeit a less frequent one, has to do with higher order functions which do not call their arguments directly, passing them instead to other higher order functions. This way, multiple levels of indirection are present between a call with a concrete function as argument and the actual higher order function which will end up using it. This is better illustrated by way of an example, like the one in Listing 3. To detect such cases and analyse them correctly in the absence of type information, we employ dataflow analysis. The current dataflow analysis we use is pretty simple and therefore not so accurate. Its details are beyond the scope of this paper. We note however that such cases account for less than 10% of the functions in the code bases we examined (cf. Sect. 4).

Another limiting factor is the fact that functions may be passed as parts of more complex data structures instead of directly as arguments. Common cases include, but

```
%% One level of indirection: it is not apparent this is a higher
%% order function since no direct call to its argument is made.
fold1(Fun, Acc, Lst) ->
  fold(Fun, Acc, Lst).

%% Two levels of indirection. The function argument has also
%% changed position.
fold2(Lst, Fun) ->
  fold1(Fun, 1, Lst).

g3() -> fold1(fun erlang:put/2, ok, [computer, error]).

g4() -> fold2([2, 3, 7], fun erlang:'*'/2).
```

Listing 3. An example of indirect higher order functions

are not limited to lists, tuples and records. In fact, most of these cases require runtime information in order to be properly resolved.

3.4 Implementation Aspects

Some aspects relating to the implementation of PURITY deserve further elaboration. The most important is the way the analysis is bootstrapped, in other words, the way we obtain the initial set of functions whose purity is predefined. This set includes all functions built-in to the ERLANG runtime system. Since these are implemented in C instead of ERLANG, they cannot be analysed. Therefore, it was necessary to extract them from the ERLANG runtime system and hard-code their purity. The values assigned were derived from their semantics, not the actual implementation.

Beyond this, it is possible to bootstrap the analysis with a more generalised mechanism, the *persistent lookup table* or PLT for short. The PLT is used to store all the information necessary to repeat the analysis as well as cached versions of the analysis' results for a given set of modules. This way, the user does not have to re-analyse every library his application depends on. The PLT also plays an important role in contexts were only one module can be analysed at a time but information regarding functions in other modules is necessary.

4 Experiences

In the course of testing our implementation diverse code bases were analysed providing some insight as to the current practices of ERLANG programmers. The applications analysed (Table 1) were primarily high profile open source projects.

Table 2 includes further information about each application. Table 3 presents the results of the analysis with the default options. Tables 4 and 5 present alternate runs of the analysis with progressively stronger purity criteria.

The columns of Table 3 labeled Pure and Impure are self-explanatory. The column labeled Undetectable represents the percentage of functions which cannot be analysed statically. These include functions like `erlang:apply(M, F, Args)` which applies function

Table 1. Brief description of the applications which were analysed

Erlang/OTP	The latest open source ERLANG distribution; among other things, it includes the bytecode and native code compilers, the standard library, static analysis tools like DIALYZER, an XML parsing library, and the Open Telecom Platform with its various networking applications
Wings3D	A subdivision modeler, used for generation of polygon models in computer graphics
CouchDB	A distributed, fault-tolerant and schema-free document oriented distributed database system
ejabberd	A server for 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	The analyzer described in this paper

Table 2. Details of analysed applications

Application	Version	Modules	Functions	LOC
Erlang/OTP	R14A	1,900	120,982	742,681
Wings3D	1.2	168	9,523	78,996
ejabberd	2.1.4	149	5,186	53,881
CouchDB	0.11.0	97	2,509	22,938
Yaws	1.88	42	1,563	19,438
Erlsom	1.2.1	18	568	9,562
ibrowse	1.6.1	7	227	2,683
purity	0.2	12	517	3,208

`F` in module `M` to some argument list of terms `Args`. Since this list can be of any length we cannot know the exact arity of the function being called at compile time. The percentage also includes functions which depend on such functions or on functions whose source code was not available during the analysis. The Limited column represents the percentage of functions which could not be conclusively analysed because of limitations in our implementation. Finally, the last column shows the CPU time required to analyse each application (in minutes/seconds), as reported by the `erlang:statistics/0` function. The tests were run on a GNU/Linux system, equipped with an Intel Core 2 Duo processor clocked at 1.6GHz and 2 GBs of RAM. Currently, our analysis only takes advantage of the second core during the information gathering stage by spawning a separate process (up to the number of available cores) for each module which is analyzed.

To better interpret the above results one should keep the following in mind. First of all, ERLANG is primarily a concurrent language and is thus expected of most applications to make extended use of concurrency primitives which render the corresponding functions impure. Furthermore, it only takes one impure function call to characterise all dependent functions as impure. Finally, although purity may initially seem as something easy to detect, reasoning about the purity of functions is not always straightforward — at least not as easy as the average programmer naïvely expects — according to our

Table 3. Analysis results with side-effects impure

Application	Pure	Impure	Undetectable	Limited	Time
Erlang/OTP	44.0%	41.4%	1.1%	13.6%	2:43
Wings3D	54.3%	34.9%	1.2%	9.6%	0:12
ejabberd	39.1%	51.2%	5.8%	4.0%	0:06
CouchDB	44.4%	44.7%	1.2%	9.7%	0:03
Yaws	44.6%	46.9%	1.1%	7.4%	0:03
Erlsom	46.0%	9.2%	0.5%	44.4%	0:02
ibrowse	44.1%	55.9%	0.0%	0.0%	0:01
purity	68.5%	20.1%	1.9%	9.5%	0:01

Table 4. Analysis results with side-effects and non-determinism impure

Application	Pure	Impure	Undetectable	Limited	Time
Erlang/OTP	37.3%	58.2%	0.6%	4.0%	2:35
Wings3D	44.6%	47.4%	1.0%	7.0%	0:10
ejabberd	33.9%	63.2%	1.6%	1.3%	0:06
CouchDB	41.8%	48.9%	0.5%	8.8%	0:03
Yaws	40.8%	52.0%	0.9%	6.3%	0:03
Erlsom	38.4%	41.5%	0.5%	19.5%	0:01
ibrowse	39.2%	60.8%	0.0%	0.0%	0:01
purity	64.4%	24.8%	1.9%	8.9%	0:01

experience. Consider for example a function like `filename:basename/1` which is part of the ERLANG standard library. This function takes a filename and returns it with the leading path component removed, e.g., `filename:basename("/usr/bin/purity")` will return `"purity"`. This function is obviously used for the value it returns and, since strings are lists in ERLANG, one would expect that this function merely performs some simple list manipulation operations. This is verified by taking a quick look at the actual source code. Most programmers would therefore consider its use consistent with programming in a purely functional style. The function is however impure as our analysis — and some more careful consideration — demonstrates. The reason has to do with portability. In order for this function to be useful across different operating systems, its behaviour needs to vary according to the character used to separate paths in each OS. It is thus dependent on the execution environment and is not referentially transparent.

The results of Table 5 in particular may appear disheartening at first. If one wishes to use the results of the analysis in contexts were exceptions cannot be regarded as pure, there is little one can gain from it. All hope is not lost however, since some of these results may be misleading. The reason so many functions appear to potentially raise exceptions is that the ERLANG compiler adds extra clauses at function definitions and `case` expressions, which raise the corresponding clause failure exception if no pattern is matched. Later optimisation passes try to remove any such clauses which are redundant, when a function is total for instance, or when it takes no arguments. Without some form of type analysis however, it is not possible to safely remove such clauses in more complex cases. An example of an ERLANG function which warrants such a clause is that of Listing 1. It is apparent from its definition that the `area` function does not cover

Table 5. Analysis results with side-effects, non-determinism and exceptions impure

Application	Pure	Impure	Undetectable	Limited	Time
Erlang/OTP	5.3%	94.6%	0.0%	0.1%	2:16
Wings3D	5.7%	94.0%	0.3%	0.1%	0:10
ejabberd	9.4%	89.8%	0.5%	0.3%	0:06
CouchDB	6.3%	92.8%	0.2%	0.7%	0:03
Yaws	7.2%	92.5%	0.1%	0.3%	0:03
Erlsom	3.9%	96.1%	0.0%	0.0%	0:01
ibrowse	6.2%	93.8%	0.0%	0.0%	0:01
purity	7.4%	91.5%	0.8%	0.4%	0:01

```
foo(42) -> ok;
foo(N) when is_integer(N) ->
  {error, N}.


bar(N) ->
  case foo(N) of
    ok -> ok;
    {error, _} -> error
  end.
```

(a) Code as written by the programmer

```
foo(42) -> ok;
foo(N) when is_integer(N) ->
  {error, N};
foo(_) -> erlang:error(badarg).

bar(N) ->
  case foo(N) of
    ok -> ok;
    {error, _} -> error;
    _ -> erlang:error(case_clause)
  end.
```

(b) Code with exceptions inserted

Fig. 2. An example where the compiler fails to remove the redundant exception raising clauses that it has inserted due to lack of type information

all possible arguments it might be called with. On the other hand the exception raising clauses will not be removed for function `bar` in the example of Figure 2. The reason is that the ERLANG compiler currently cannot determine that the pattern matching on the return value of the call to `foo` is complete, since it does not keep any information regarding `foo`'s return value.

Furthermore, these percentages do not account for the masking of exceptions by other functions. Consider the example in Listing 4 where an exception is raised by one function but is later masked in the body of another. With a more sophisticated analysis it is possible that some of these cases can be detected.

```
foo(X) ->
  throw(X).

bar() ->
  try foo(42) of
    Val -> {ok, Val}
  catch
    throw:E -> {error, E}
  end.
```

Listing 4. An example of exception masking

Table 6. Analysis results with side-effects impure and termination analysis

Application	Pure	Impure	Undetectable	Limited
Erlang/OTP	23.1%	74.8%	0.5%	1.7%
Wings3D	30.5%	67.6%	0.8%	1.1%
ejabberd	26.6%	69.3%	2.3%	1.7%
CouchDB	30.2%	65.0%	0.4%	4.4%
Yaws	36.0%	61.3%	0.7%	2.0%
Erlsom	29.2%	57.4%	0.4%	13.0%
ibrowse	36.1%	63.9%	0.0%	0.0%
purity	50.9%	45.8%	1.4%	1.9%

For completeness we include one more table. Table 6 presents the results of Table 3 when these are combined with a simple termination analysis we have also developed (whose details are beyond the scope of this paper). In effect, the first column of the table shows how many functions are found both pure and terminating. Why this information is interesting is described in the next section.

5 Extending Erlang with User-Defined Guards

The original motivation for our analysis was to extend the ERLANG language with support for user-defined guards. Besides being a very popular request by users, such an extension is important since it increases the language's expressiveness and allows for more compact and descriptive code.

We mentioned in Section 2 that guard expressions are currently limited to a predefined set of built-in functions. The reason for this is that the ERLANG specification requires that guard expressions are an extension of pattern matching. As such, functions used in guards should have no observable side-effects and should complete in bounded time. Notice that these prerequisites do not mention determinism. In fact, valid ERLANG guard expressions include the `erlang:node/0` and `erlang:node/1` BIFs, which depend on the execution environment. Specifically, the former returns the name of the current ERLANG node,[1] while the latter returns the name of the node a specific process belongs to. It is possible to change this node name from within the ERLANG runtime system by calling functions `start/1` and `stop/0` of the `net_kernel` module. The example in Figure 3 shows an excerpt from a session in the ERLANG shell, illustrating how a guard expression might succeed on one call and fail on another.

Another aspect of ERLANG guards which has not been discussed yet, is that any exceptions which may be raised as part of a guard test are caught and silently converted to the value `false`. That is to say that functions that raise an exception during normal execution, will not do so when used in a guard context.

With these issues in mind, it should now be clear why we chose to support multiple criteria of purity in our analysis. The guard in the previous example is not a referentially transparent function and a more strict analysis would reject it. Other valid guard tests,

[1] Taken from the ERLANG manual "A node is an executing Erlang runtime system which has been given a name"[5, Chapter 12].

```
1> F = fun () when node() =:= nonode@nohost -> error;
           () -> {ok, node()}
       end.
#Fun<erl_eval.6.13229925>
2> F().
error
3> net_kernel:start([test@localhost]).
{ok,<0.36.0>}
(test@localhost)4> F().
{ok,test@localhost}
(test@localhost)5> net_kernel:stop().
ok
6> F().
error
```

Fig. 3. Example of a non-deterministic guard expression

e.g., functions like `erlang:node/1` and `erlang:length/1`, raise exceptions when called with invalid arguments outside of guard contexts. Obviously, we did not want to break any existing code with our extension. As can be seen from Table 6, there is a fair number of functions in existing ERLANG programs that our analyses classify as both pure and terminating and therefore it could enable their employment as a guard.

In our opinion, one of the biggest advantages that such a language extension has to offer, is the fact that it enables user-defined tests for abstract data types in guards. The problem with ADTs in ERLANG is that their structural information is exposed, as the language allows inspection through pattern matching and primitive type tests. Allowing arbitrary type tests as guards can help make code cleaner and could even discourage programmers from breaking ADT contracts. The example in Figure 4 illustrates the benefits of such an approach.

Prototype Implementation. To this end, we developed a proof of concept implementation on top of the ERLANG compiler. Two distinct aspects of the compilation process have been altered, while a third modification has been identified in the runtime system of Erlang/OTP. First of all, a compiler pass performing purity and termination analysis is placed in the compiler front-end, just after the pass which converts ERLANG source to CORE ERLANG. Additionally, errors regarding illegal guard expressions are silenced until the purity of the functions in question can be verified by looking up their values. Second, the compiler back-end needs to be changed, specifically the code generation stage up to the point where bytecode is produced. This was the trickiest part, since this compiler phase heavily relied on the assumption that only built-in functions might be called from within guard expressions. BIFs differ significantly from a regular ERLANG function call with respect to the bytecode that needs to be generated.

A third aspect we identified but have not implemented yet has to do with the ERLANG loader. As mentioned earlier, ERLANG is unique in that it supports loading new code while the system is still running. It should be evident that a check must be placed at this stage, to verify that the same properties hold for the newly loaded code as the code which was previously analyzed, specifically with regard to its purity and termination

```
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.
```

(a) Custom tests not allowed as guards

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

(b) When user-defined guards are allowed

Fig. 4. Two ways of writing a function that operates on different term representations when user-defined tests are forbidden as guards 4(a) and when they are allowed 4(b). The code is not only more succinct, but it is also significantly more clear. Writing the code of Figure 4(b) is something currently not possible in ERLANG.

characteristics, and take some appropriate action (e.g. refuse the re-loading or re-compilation of modules that depend on this code) if there is a difference. Other engineering issues not addressed in our current prototype have to do with optional user annotations of pure functions. Such annotations would make the programmer's intentions more explicit and could be further used by the code loader.

6 Related Work

Purity is a fundamental property of programming components, regardless of language. In this respect, it is a bit surprising that in the literature it is hard to locate descriptions of analyses that detect purity or papers that report statistical data about the purity aspects of programs. In particular, we are not aware of any published such works in the context of dynamically typed functional languages. Still, being able to determine side-effect freeness is important for optimisation, automatic parallelization, and for program transformation tools such as refactoring editors. No doubt many such systems probably contain analysis components that determine purity properties of programs of different degrees.

Statically typed languages on the other hand, usually encode purity information in their type system. One approach which may be employed in languages with imperative features is a *type and effect system* (see, e.g., Lucassen and Gifford's paper [6] or Chapter 3 in Pierce's book [8]). Such a system extends a traditional type system with information about how values are computed and the possible effects that expressions can have (such as their side-effects or the set of memory regions that may be modified as a result of their evaluation).

Conversely, practical constraints — such as efficient I/O operations — compel many purely functional programming languages to allow for impure operations, without violating their pure semantics. Different approaches have been explored, the most notable

of which are the uniqueness type system of Clean and the monadic approach of Haskell. The equivalence between monads and effect systems has also been ascertained [12].

Clean is a general purpose, strongly typed, pure and lazy functional programming language. Clean handles side-effects and non-determinism by means of a *uniqueness typing* system [9, ch. 9]. This extends a traditional type system by allowing the user to specify that a given argument to a function is unique. Such an annotation guarantees the function will have private access to the argument, therefore destructively updating it will not violate the semantics of the function during the execution of the program. Besides side-effects, uniqueness typing can be used to convert pure operations to mutable state transformations without violating the pure semantics of the operation. With a uniqueness guarantee it is trivial to verify referential transparency, as the function will never be called with the same argument. Furthermore, any side-effects of the function will never influence another function in an unforeseen manner [11].

Haskell is similar to Clean in many respects. It is purely functional, strongly typed and also features non-strict evaluation. However, Haskell takes a different approach with respect to purity, utilizing the more general concept of *monads*. Like Clean, this information is reflected in the type system and can be automatically inferred by way of a type inference scheme. Besides side-effects, monads can be used to express more general, and not necessarily impure, computations [7].

BitC was developed as a systems programming language with the goal of supporting formal verification. Unlike the languages previously mentioned, BitC is not purely functional. It does however support user level type annotations regarding the purity of functions, by means of an effect type system [10, ch. 10]. Such annotations associate expressions with an effect type variable which can have a value of *pure*, *impure* or *unfixed*. By verifying that certain parts of a program are pure, the BitC compiler can safely perform certain kinds of optimisations, like automatic parallelization.

7 Concluding Remarks

In this paper, having presented the defining properties of impure functions according to three independent and progressively weaker criteria (presence of side-effects, dependency on the environment of execution and possibility of raising an exception), we described the design and implementation of a parameterized static analysis for determining such properties in the context of ERLANG. Our analysis is relatively simple, but it has very important consequences both for the optimisation and, more importantly, for the expressivity of ERLANG programs. As a direct result of the existence of our analysis, we are currently enhancing the ERLANG language with user-defined guards and have already developed a suitable patch to the compiler which can form the basis of the final implementation in Erlang/OTP.

In the course of testing our implementation, we have analyzed diverse code bases of significant size, providing concrete data regarding the current practices of ERLANG programmers with respect to purity. The percentage of functions classified as pure by

our analysis ranges on average between 30% and 50% for the applications we examined. This percentage is significant considering that ERLANG is primarily a concurrent language. To the best of our knowledge, this is the first time that such data is reported in the literature, not only for ERLANG, but for any functional language where purity of functions is not captured by the type system.

Acknowledgements. We thank Patrick Maier and Phil Trinder for their detailed suggestion on how to show a run of the analysis on an example, which has clearly improved the presentation aspects of our work.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Barklund, J., Virding, R.: Erlang 4.7.3 reference manual (February 1999), `http://www.csd.uu.se/ftp/mirror/erlang/download/erl_spec47.ps.gz`
3. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.O., Pettersson, M., Virding, R.: Core Erlang 1.0 language specification. Tech. Rep. 030, Information Technology Department, Uppsala University (November 2000)
4. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlangs exception handling revisited. In: Proceedings of the ACM SIGPLAN Workshop on Erlang, pp. 16–26. ACM (2004)
5. Ericsson, A.B.: Erlang Reference Manual Users Guide, version 5.8 (June 2010), `http://www.erlang.org/doc/reference_manual/users_guide.html`
6. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, pp. 47–57. ACM, New York (January 1988)
7. Newbern, J.: All about monads, `http://www.haskell.org/all_about_monads/`
8. Pierce, B.C. (ed.): Advanced Topics in Types and Programming Languages. MIT Press, Cambridge (2005)
9. Plasmeijer, R., van Eekelen, M.: Clean Language Report, version 2.1 (November 2002), `http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf`
10. Shapiro, J., Sridhar, S., Doerrie, M.S.: The origins of the BitC programming language (April 2008), `http://www.bitc-lang.org/docs/bitc/bitc-origins.html`
11. de Vries, E., Plasmeijer, R., Abrahamson, D.M.: Uniqueness typing simplified. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 201–218. Springer, Heidelberg (2008)
12. Wadler, P.: The marriage of effects and monads. ACM SIGPLAN Notices 34(1), 63–74 (1999)