Science of Computer Programming 110 (2015) 104–118

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A zoom-declarative debugger for sequential Erlang programs

Rafael Caballero [a,*], Enrique Martin-Martin [a], Adrián Riesco [a], Salvador Tamarit [b]

[a] Dpto. de Sistemas Informáticos y Computación, Fac. Informática, Universidad Complutense de Madrid, C/ Profesor José García Santesmases, 9, 28040 Madrid, Spain
[b] Babel Research Group, Fac. Informática, Universidad Politécnica de Madrid, Campus de Montegancedo, s/n. 28660 Boadilla del Monte, Spain

ARTICLE INFO

Article history:
Received 18 March 2015
Received in revised form 24 June 2015
Accepted 30 June 2015
Available online 8 July 2015

Keywords:
Erlang
Zoom debugging
Declarative debugging

ABSTRACT

We present a declarative debugger for sequential Erlang programs. The tool is started when a program produces some unexpected result, and proceeds asking questions to the user about the correctness of some subcomputations until an erroneous program function is found. Then, the user can refine the granularity by zooming in the function, checking the values bound to variables and the `if/case/try-catch` branches taken during the execution. We show by means of an extensive benchmark that the result is a usable, scalable tool that complements already existing debugging tools such as the Erlang tracer and Dialyzer. Since the technique is based on a formal calculus, we are able to prove the soundness and completeness of the approach.

1. Introduction

Erlang [12] is a programming language that combines the elegance and expressiveness of functional languages (higher-order functions, lambda abstractions, single assignments), with features required in the development of scalable commercial applications (garbage collection, built-in concurrency, and even hot-swapping). The language is used as the base of many fault-tolerant, reliable software systems. The development of these systems is a complicated process where tools such as discrepancy analyzers [23], test-case generators [27], or debuggers play an important rôle. In the case of debuggers, Erlang provides a useful trace debugger including different types of breakpoints, stack tracing, and other features. However, debugging a program is still a difficult, time-consuming task—according to a National Institute of Standards and Technology (NIST) study, software engineers spend 70–80% of their time testing and debugging [31]. Therefore alternative or complementary debugging tools are still needed.

Taking advantage of the declarative nature of the sequential subset of Erlang, in this paper we present a new debugger based on the general technique known as *declarative debugging* [33]. Also known as *declarative diagnosis* or *algorithmic debugging*, this technique abstracts the execution details, which may be difficult to follow in declarative languages, to focus on the validity of the results. This approach was first proposed in the logic paradigm [25,37], where debugging programs including features like backtracking using a traditional trace debugger can be really difficult. Soon, declarative debugging was applied to functional [26,29] and multi-paradigm [6,24] languages. Declarative debuggers of non-strict functional languages display the terms evaluated to the point required by the actual computation, something very useful in these languages which include the possibility of representing infinite data structures. More recently, the same principles have been applied

* Corresponding author.
E-mail addresses: rafacr@ucm.es (R. Caballero), emartinm@ucm.es (E. Martin-Martin), ariesco@fdi.ucm.es (A. Riesco), stamarit@babel.ls.fi.upm.es (S. Tamarit).

http://dx.doi.org/10.1016/j.scico.2015.06.011
0167-6423/

```
1 mod(X,Y)-> (X  rem Y + Y) rem Y.

2 to_pos(L) when L >= $ A, L =< $Z ->L - $A.
3 from_pos(N)-> mod(N, 26)+ $A.

4 encipher(P, K)->from_pos(to_pos(P)+ to_pos(K)).
5 decipher(C, K)->from_pos(to_pos(C)- to_pos(K)).

6 cycle_to(N, List) when length(List)>= N ->List;
7 cycle_to(N, List)->lists:append(List,cycle_to(N - length(List),List)).

8 normalize(Str)->toupper(filter(fun isalpha/1, Str)).

9 crypt(RawText, RawKey, Func)->
10    PlainText = normalize(RawText),
11    lists:zipwith(Func, PlainText,
12                  cycle_to(length(PlainText), normalize(RawKey))).

13 encrypt(Text, Key)->crypt(Text, Key, fun encipher/2).
14 decrypt(Text, Key)->crypt(Text, Key, fun decipher/2).
```

Fig. 1. Erlang code implementing Vigenère cipher.

to object-oriented [7,22] programming languages. The basic idea of this scheme consists of asking questions to the user about the validity of the subcomputations associated to a computation that has produced an error until an erroneous fragment of code is located.

Our tool provides features such as trusting, support for higher-order functions and built-ins, and "don't know" answers. In fact, checking the comparison from [30], we can see that our debugger has most of the features in state-of-the-art tools, although we still lack some elements such as a graphical user interface, or more navigation strategies. Typical declarative debuggers in functional programming usually look for incorrect functions. However, in real-world programs, functions can be quite intricate. Thus, detecting that a function is erroneous, although helpful, still leaves to the user the problem of finding *where* is the error inside the body function. For this reason we allow the user to employ declarative debugging also *inside* an erroneous function in order to detect the precise piece of code that caused the bug. We call this feature *zoom debugging* and constitutes, to the best of our knowledge, the first work where such a feature is described. The present work extends and completes the results in [8], where we introduced the foundations of the debugger, and in [9], where a short explanation of the tool was presented, by:

- Comparing in detail the different debugging techniques that can be used in Erlang, focusing on their strengths and flaws.
- Giving a detailed description of *zoom debugging*, which was just succinctly presented in [9]. This feature is not present in any other declarative debugger.
- Compiling a study of the applicability of the system to real programs. We have applied our debugger to a wide range of small-medium applications *developed by others* and real-world projects in the Erlang community obtained from *GitHub*. In all these cases our tool has been able to find the errors using a small number of questions. This gives us confidence on the usability and potential of the tool.

The rest of the paper is organized as follows: Section 2 introduces Erlang by means of a motivating example. Section 3 describes different techniques to debug Erlang and the similarities with our approach. Section 4 describes the main features of our tool, while Section 5 focuses on zoom debugging. Section 6 recounts the main theoretical results. Section 7 presents the experimental results obtained when using our tool, while Section 8 concludes and presents the future and ongoing work.

2. Motivating example

We start introducing a running example taken from the Erlang community. It is important to remark that the bug was already in the program and has not been introduced by the authors.

Fig. 1 presents a *Vigenère cipher* [5] written in Erlang, extracted from the programming chrestomathy *Rosetta Code*.[1] The program exports the functions `encrypt/2` and `decrypt/2` that, given a text and a key, encipher or decipher it, respectively. The Vigenère cipher is a simple and well-known method for encrypting alphabetic text by adding *modulo 26*

[1] http://rosettacode.org/mw/index.php?title=Vigen%C3%A8re_cipher.