Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Casper: Automatic tracking of null dereferences to inception with causality traces

Benoit Cornu[a], Earl T. Barr[b], Lionel Seinturier[a], Martin Monperrus[a,*]

[a] Centre de Recherche en Informatique Signal et Automatique de Lille Université de Lille 1 59655 Villeneuve d'Ascq Cedex France
[b] Department of Computer Science, University College London, Gower Street London, WC1E 6BT, UK

ARTICLE INFO

Article history:
Received 11 January 2016
Revised 16 August 2016
Accepted 17 August 2016
Available online 18 August 2016

Keywords:
Debugging
Null pointer
Causality analysis

ABSTRACT

Fixing a software error requires understanding its root cause. In this paper, we introduce "causality traces", crafted execution traces augmented with the information needed to reconstruct the causal chain from the root cause of a bug to an execution error. We propose an approach and a tool, called Casper, based on code transformation, which dynamically constructs causality traces for null dereference errors. The core idea of Casper is to replace `nulls` with special objects, called "ghosts", that track the propagation of the nulls from inception to their error-triggering dereference. Causality traces are extracted from these ghosts. We evaluate our contribution by providing and assessing the causality traces of 14 real null dereference bugs collected over six large, popular open-source projects.

1. Introduction

Null pointer dereferences are frequent errors that cause segmentation faults or uncaught exceptions. Li et al. found that 37.2% of all memory errors in Mozilla and Apache are null dereferences (Li et al., 2006). Kimura et al. (2014) found that there are between one and four null checks per 100 lines of code on average. Problematic null dereferences are daily reported in bug repositories, such as bug MATH#305.[1] A null dereference occurs at runtime when a program tries to read memory using a field, parameter, or variable that points to "`null`", i.e. nothing. The terminology changes depending on the language, in this paper, we concentrate on the Java programming language, where a null dereference triggers an exception called NullPointerException, often called "NPE".

Just like any bug, fixing null dereferences requires understanding their root cause, a process that we call *null causality analysis*. At its core, this analysis is the process of connecting a null dereference, where the fault is activated and whose symptom is a null pointer exception, to its root cause, usually the initial assignment of a null value, by means of a *causality trace* — the execution path the null took through the code from its inception to its dereference.

The literature offers different families of techniques to compute the root cause of bugs, mainly program slicing, dataflow analysis, or spectrum-based fault-localization. However, they have been little studied and evaluated in the context of identifying the root cause of null dereferences (Hovemeyer and Pugh, 2004; Bond et al., 2007; Wang et al., 2013). Those techniques are limited in applicability (Hovemeyer and Pugh (2004) is an intra-procedural technique) or in accuracy (program slicing results in large sets of instructions (Binkley and Harman, 2004)). The fundamental problem not addressed in the literature is that the causality trace from null inception to the null symptom is missing. This is the problem that we address in this paper. We propose a causality analysis technique that uncovers the inception of null variable bindings that lead to errors along with the causal explanation of how null flowed from inception to failure during the execution. While our analysis may not report the root cause, it identifies the null inception point with certainty, further localizing the root cause and speeding debugging in practice.

Let us consider a concrete example. Listing 1 shows a null dereference stack trace which shows that the null pointer exception happens at line 88 of BisectionSolver. Let's assume that a perfect fault localization tool suggests that this fault is located at line 55 of UnivariateRealSolverImpl (which is where the actual fault lies). However, the developer is left clueless with respect to the relation between line 55 of UnivariateRealSolverImpl and line 88 of BisectionSolver where the null dereference happens.

What we propose is a *causality trace*, as shown in Listing 2. In comparison to Listing 1, it contains three additional pieces of information. First, it gives the exact name, here f, and kind, here parameter (local variable or field are other possibilities), of the

* Corresponding author.
E-mail address: martin.monperrus@univ-lille1.fr (M. Monperrus).
[1] https://issues.apache.org/jira/browse/MATH-305.

```
Exception in thread "main" java.lang.NullPointerException
    at [..].BisectionSolver.solve(88)
    at [..].BisectionSolver.solve(66)
    at ...
```

Listing 1. The standard stack trace of a real null dereference bug in Apache Commons Math.

```
Exception in thread "main" java.lang.NullPointerException
Dereferenced parameter "f" // symptom
    at [..].BisectionSolver.solve(88)
    at [..].BisectionSolver.solve(66)
    at ...
Parameter f bound to field "f2"
    at [..].BisectionSolver.solve(66)
Field "f2" set to null
    at [..].UnivariateRealSolverImpl.<init>(55) //
         cause
```

Listing 2. What we propose: a causality trace, an extended stack trace that contains the root cause.

variable that holds null.[2] Second, it explains the inception of the null binding to the parameter, the call to solve at line 66 with field f2 passed as parameter. Third, it gives the root cause of the `null` dereference: the assignment of null to the field f2 at line 55 of class UnivariateRealSolverImpl. Our causality traces contain several kinds of causal links, of which Listing 2 shows only three: the name of the wrongly dereferenced variable, the flow of a null binding through parameter bindings, and null assignment. Section 2.2 presents the concept of null causality trace.

We present Casper, a tool that transforms Java programs to capture causality traces and facilitate the fixing of null deferences.[3] Casper takes as input the program under debug and a main routine that triggers the null dereference. It first instruments the program under debug by replacing `null` with "ghosts" that are shadow instances responsible for tracking causal information during execution. To instrument a program, Casper applies a set of 11 source code transformations tailored for building causal connections. For instance, x = y is transformed into o = assign(y), where method assign stores an assignment causal links in a null ghost (Section 2.3). Section 2.4 details these transformations.

Compared to the related work, Casper is novel along three dimensions. First, it collects the complete causality trace from the inception of a null binding to its dereference. Second, it identifies the inception of a null binding with *certainty*. Third, Casper is lightweight and easily deployable, resting on transformation rather than replacing the Java virtual machine, a la Bond et al. (2007). The first two properties strongly differentiate Casper from the related work (Sinha et al., 2009; Bond et al., 2007), which tentatively labels root causes with suspiciousness values and does not collect causality traces nor identifies null inception points with certainty.

We evaluate our contribution Casper by providing and assessing the causality traces of 14 real null dereference bugs collected over six large, popular open-source projects. We collected these bugs from these project's bug reports, retaining those we were able to reproduce. Casper constructs the complete causality trace for 13 of these 14 bugs. For 11 out of these 13 bugs, the causality trace contains the location of the actual fix made by the developer.

To sum up, our contributions are:

- The definition of causality traces for null dereference errors from null inception to its dereference and the concept of "ghost" classes, which replace `null`, collect causality traces, while being otherwise indistinguishable from `null`.
- A set of code transformations that inject null ghosts and collect causality traces of null dereferences.
- Casper, an Java implementation of our technique.
- An evaluation of our technique on 14 real null dereference bugs collected over 6 large open-source projects.

The remainder of this paper is structured as follows. Section 2 presents our technical contribution. Section 3 gives the results of our empirical evaluation. Section 4 discusses the limitations of our approach. Sections 5 and 6 respectively discusses the related work and concludes. Casper and our benchmark can be downloaded from https://github.com/Spirals-Team/casper.

2. Debugging nulls with Casper

Casper tracks the propagation of a `null` binding during application execution in a causality trace. A *`null` dereference causality trace* is a sequence of program elements (AST nodes) traversed during execution from the source of the `null` to its erroneous dereference.

2.1. Overview

We replace `null`s with objects whose behavior, from the application's point of view, is same as `null`, except that they store a causality trace, defined in Section 2.2. We call these objects *null ghosts* and detail them in Section 2.3. Casper rewrites the program under debug to use null ghosts and to store a `null`'s causality trace in those null ghosts, (see Section 2.4). We instantiated Casper's concepts in Java and therefore tailored our presentation in this section to Java (Section 2.5).

Casper makes minimal assumptions on the application under debug, in particular, it does not assume a closed world where all libraries are known and manipulable. Hence, a number of techniques used in Casper comes from this complication.

2.2. Null dereference causality trace

To debug a complex null dereference, the developer has to understand the history of a null binding from its inception to its problematic dereference. When a variable is set to `null`, we say that a null binding is created. When clear from context, we drop "binding" and say only that a null is created. This is a conceptual view that abstracts over the programming language and the implementation of the virtual machine. In Java, there is a single `null` value to which variables are bound without creating a new `null` values.

To debug a null dereference, the developer has to know the details of the `null`'s propagation, i.e. why and when each variable became null at a particular location. We call this history the "null causality trace" of the null dereference. Developers read and write source code. Thus, source code is the natural medium in which developers reason about programs for debugging. In particular, a `null` propagates through assignments and method return values. This is why Casper defines causal links in a null causality trace in terms of traversed program elements and their actual location in code, defined as follows and presented in Table 1.

Definition 2.1. A null dereference causality trace is the temporal sequence of program elements (AST nodes) traversed by a dereferenced `null`.

[2] if the dereference is the result of a method invocation, we give the code expression that evaluates to null.

[3] We have named our tool Casper, since it injects "friendly" ghosts into buggy programs.