CHECKERS: Multi-modal Darwinian API Optimisation

Santanu Kumar Dash
University of Surrey
United Kingdom
s.dash@surrey.ac.uk

Fan Wu, Michail Basios, Lingbo Li
Leslie Kanthan
Turing Intelligence Technology, United Kingdom
{fan,mike,lingbo,leslie}@turintech.ai

ABSTRACT

Advent of microservices has increased the popularity of the API-first design principles. Developers have been focusing on concretising the API to a system before building the system. An API-first approach assumes that the API will be correctly used. Inevitably, most developers, even experienced ones, end-up writing sub-optimal software because of using APIs incorrectly. In this paper, we discuss an automated approach for exploring API equivalence and a framework to synthesise semantically equivalent programs. Unlike existing approaches to API transplantation, we propose an *amorphous* or formless approach to software translation in which a single API could potentially be replaced by a synthesised sequence of APIs which ensures type progress. Our search is guided by the non-functional goals for the software, a type-theoretic notion of progress, the application's test suite and an automatic multi-modal embedding of the API from its documentation and code analysis.

ACM Reference Format:
Santanu Kumar Dash, Fan Wu, Michail Basios, Lingbo Li, and Leslie Kanthan. 2020. CHECKERS: Multi-modal Darwinian API Optimisation. In *IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387940.3392173

1 INTRODUCTION

The popularity of cloud based applications and the need for scalability and integration with various applications has lead to a microservice based approach when developing applications. Terms like API-first development and API-first design are becoming increasingly popular among developers [6]. This has lead to extensive usage of external APIs in applications. Consequently, the selection of the right API is central to achieving the best in efficiency, responsiveness, scalability, throughput, and memory usage [1, 3, 7, 11].

There is a need for automated API optimisation tools to fully exploit API first design principles. Settling on an API is difficult when requirements are continuously evolving. Consider Android's `SurfaceView` and `TextureView` classes which can both be used to create dedicated drawing surfaces. `SurfaceView` does not allow frames to be animated, transformed or scaled but `TextureView` does, although at lower frame throughput. Choosing one over the other is hard if there is a lack of clarity on whether animated frames

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392173

```
1 public ByteBuffer getFrame() {
2     mPixelBuf.rewind();
3     GLES20.glReadPixels(0, 0, mWidth, mHeight,
          format, type, mPixelBuf);
4     return mPixelBuf;
5 }
```

Listing 1: `glReadPixels` is used to read a block of pixels from a frame. Here, `format` ∈ {`GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT_*`} and `type` ∈ {`GL_ALPHA`, `GL_RGB`, `GL_RGBA`}.

might be required. Even if the right API can be identified, actual parameters need to be tuned to meet non-functional requirements. For example, a common mistake in `OpenGL` programming is the usage of incorrect types for pixel depths. In Listing 1, precious CPU cycles are used in converting depth formats for pixels from normalised integer values to floating points if `GL_FLOAT` is used instead of `GL_UNSIGNED_BYTE` or `GL_UNSIGNED_SHORT_*` to represent depth buffer precision.

Additionally, migrating APIs across versions while optimising their performance can take extensive manual work and testing. Consider Android applications as an example. Android's codebase is fast-moving; APIs are frequently added and deprecated. It is not uncommon for developers to have multiple APIs to achieve the same task. This inevitably leads to mistakes in choosing the right API. Even when the API for a subsystem is relatively stable, there are parameters that need to be set properly to make the app faster and reduce energy consumption. Take the case of the palette API as an example, it was shown in [8] that altering the color composition of the GUI can lead to significant energy savings.

In this paper, we propose a framework to identify API calls, automatically find substitutions for them and test the substituted software for non-functional requirements. For API calls, we replace them with either a single API or a sequence of API calls that are semantically equivalent to the original call. As we permit sequences of APIs as substitutions, our search process is more involved that finding APIs with the same type signature. To ameliorate the search process, in a first, we aim to harness API documentation to guide the search.

2 SEARCHING FOR EQUIVALENT APIS

For every API call site, there are two ways in which potential replacements can be identified without causing type errors. We call these *Singular* and *Compositional* replacements.

Definition 2.1. The *type environment* Γ for a program is the mapping of terms in a grammar to their type where $\Gamma(x)$ returns the type for the terms x. If x is a method, $\Gamma^i(x)$ returns a sequence of its input types and $\Gamma^o(x)$ returns a sequence of its output types.

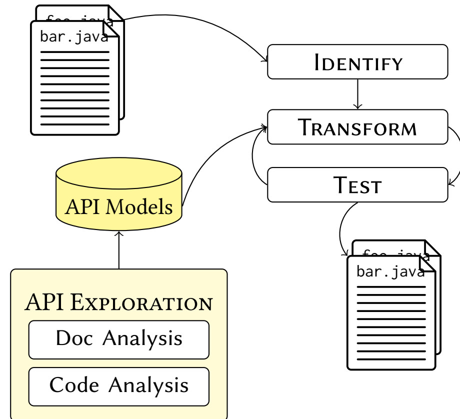

Figure 1: Overview of Checkers.

Definition 2.2. *Singular Replacement* is the replacement of a single API call f with g such that $\Gamma(f) = \Gamma(g)$

The `ImageReader` class in Android contains examples of candidates for singular transplantation. It contains two methods for reading images: `acquireLatestImage()` and `acquireNextImage()` with identical type signatures. The former gets the next `Image` from the `ImageReader` queue but applies the `close()` method on all instances of `Image` that are open. However, the `acquireNextImage()` does not close older instances of `Image`. This has an impact on memory consumption and Android's developer notes recommend using `acquireNextImage()` only for background/batch processing.

Definition 2.3. *Compositional Replacement* is the replacement of a single API call f with a sequence of type-correct API calls $g_1(g_2(g_3(\cdots g_n(x_1, x_2, \cdots, x_k)\cdots)))$ such that $\forall k.\Gamma^i(g_k) = \Gamma^o(g_{k+1}) \land \Gamma^i(g_n) = \Gamma^i(f) \land \Gamma^o(g_1) = \Gamma^o(f)$

Examples of compositional replacement, which has the same theoretical underpinnings as type-directed program synthesis [10], can be found in the `RecyclerView` class which displays lists, that can be dynamically updated, in a constrained widget. If `RecyclerView` needs to display a list that is re-fetched from the network or the database upon update, there are three ways to achieve this. The first is through a `ListAdapter` API which diffs the lists on a background thread unblocking the main thread. The second is through the `AsyncListDiffer` which does the same task through a callback. Finally, there is the low-level `DiffUtil` class which achieves the same task on a background thread. Each of the three techniques use a combination of API calls but are semantically equivalent.

Type-correctness of the replacement does not automatically imply semantic equivalence with the original API. Two methods with identical type signatures could be doing different tasks. Therefore, we rely on the program's test suite to establish a weak form of semantic equivalence for candidate replacements with the same type signature. Our framework for optimisation is described next.

3 MULTI-STAGE API OPTIMISATION

An overview of Checkers is shown in Figure 1. Checkers uses combination of code and documentation for optimisation. It builds on recent work in dual-channel research which has shown to benefit standard forms of analysis by drawing signal from both the human-human or natural language channel and human-machine or programming language channel in software [4, 5, 9].

Checkers aims to identify candidates for singular and compositional replacements. For this, it relies on a one-time extraction of an API's type signatures and parsing of its documentation [9] to build API models. This process is shown in yellow in Figure 1. Each API's representation has two components: a type signature and a vector embedding derived from the documentation for API. While the type signature helps identify type-correct replacements, the vector embedding guides the search process for compositional replacements by using embedding to group together APIs with related documentation.

We use static analysis and program transformation to produce a candidate program which improves upon the original program. A key point in our approach is that for any replacement, we also try to tune individual parameters to the API wherever possible and auto-parsing of the API documentation helps identify tunable parameters such as flags. The three main stages in our rewriting are Identify, Transform and Test. The Identify stage parses the source to identify locations for target APIs. The Transform stage searches for candidate replacement amongst API models and the Test phase runs unit and integration tests on the rewritten code to sanity check the rewriting. We adopt a similar approach to [2] which uses test-driven optimisation and loops until it reaches a desired level of improvement or times out.

REFERENCES

[1] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2017. Optimising Darwinian Data Structures on Google Guava. In *Search Based Software Engineering*, Tim Menzies and Justyna Petke (Eds.). Springer International Publishing, Cham, 161–167.

[2] Michail Basios, Lingbo Li, Fan Wu, Leslie Kanthan, and Earl T. Barr. 2018. Darwinian Data Structure Selection. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. 118–128.

[3] Bobby R. Bruce, Justyna Petke, and Mark Harman. 2015. Reducing Energy Consumption Using Genetic Improvement *(GECCO '15)*. Association for Computing Machinery, New York, NY, USA, 1327–1334.

[4] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. 2020. A Theory of Dual Channel Constraints. In *International Conference on Software Engineering (ICSE), New Ideas and Emerging Results (NIER)*. To appear.

[5] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2018. RefiNym: using names to refine types. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, 2018*. 107–117.

[6] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. In *Present and ulterior software eng.* Springer, 195–216.

[7] William B. Langdon, Westley Weimer, Christopher Timperley, Oliver Krauss, Zhen Yu Ding, Yiwei Lyu, Nicolas Chausseau, Eric Schulte, Shin Hwei Tan, Kevin Leach, and et al. 2019. The State and Future of Genetic Improvement. *SIGSOFT Softw. Eng. Notes* 44, 3 (Nov. 2019), 25–29.

[8] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2018. Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps. *ACM Transactions on Software Engineering and Methodologies* 27, 3, Article Article 14 (Sept. 2018), 47 pages.

[9] Profir-Petru Pârţachi, Santanu Kumar Dash, Christoph Treude, and Earl T. Barr. 2020. POSIT: Simultaneously Tagging Natural and Programming Languages. In *International Conference on Software Engineering (ICSE)*. To appear.

[10] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 522–538.

[11] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. 2015. Deep Parameter Optimisation *(GECCO '15)*. ACM, 1375–1382.