Science of Computer Programming 105 (2015) 145–170

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Systematic derivation of correct variability-aware program analyses [☆]

Jan Midtgaard [a,1], Aleksandar S. Dimovski [b], Claus Brabrand [b,*], Andrzej Wąsowski [b]

[a] DTU Compute, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
[b] IT University of Copenhagen, 2300 Copenhagen S, Denmark

ARTICLE INFO

Article history:
Received 27 February 2014
Received in revised form 13 April 2015
Accepted 14 April 2015
Available online 27 April 2015

Keywords:
Software Product Lines
Software variability
Verification
Static analysis
Abstract interpretation

ABSTRACT

A recent line of work *lifts* particular verification and analysis methods to Software Product Lines (SPL). In an effort to generalize such case-by-case approaches, we develop a systematic methodology for lifting single-program analyses to SPLs using abstract interpretation. Abstract interpretation is a classical framework for deriving static analyses in a compositional, step-by-step manner. We show how to take an analysis expressed as an abstract interpretation and lift each of the abstract interpretation steps to a family of programs (SPL). This includes schemes for lifting domain types, and combinators for lifting analyses and Galois connections. We prove that for analyses developed using our method, the soundness of lifting follows by construction. The resulting *variational abstract interpretation* is a conceptual framework for understanding, deriving, and validating static analyses for SPLs. Then we show how to derive the corresponding variational dataflow equations for an example static analysis, a constant propagation analysis. We also describe how to approximate variability by applying variability-aware abstractions to SPL analysis. Finally, we discuss how to efficiently implement our method and present some evaluation results.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

The methodology of *Software Product Lines* (SPLs) [1] enables systematic development of a *family* of related programs, known as *variants*, from a common code base by maximizing reuse in order to decrease development cost and time-to-market. Each variant in an SPL is specified in terms of *features* selected for that particular variant. The SPL method has grown in popularity over the last 20 years, especially in the domain of embedded systems, including safety critical systems with stringent quality requirements on produced code.

While program families can be implemented using domain-specific languages and general-purpose model transformation [2], often it is possible to use simpler methods that are more easily amenable to testing and analysis. The most popular [3] implementation method in practice relies on a simple form of two-staged computation in preprocessor style: the programming language used (often C) is enriched with the ability to express simple compile-time computations (often

☆ Supported by *The Danish Council for Independent Research* (grant no. 0602-02327B) under the Sapere Aude scheme, project VARIETE.
* Corresponding author.
E-mail address: brabrand@itu.dk (C. Brabrand).
1 A major part of this work was carried out while the author was at Dept. of Computer Science, Aarhus University, 8200 Aarhus N, Denmark.

http://dx.doi.org/10.1016/j.scico.2015.04.005
0167-6423/© 2015 Elsevier B.V. All rights reserved.

C preprocessor), e.g., it can be enriched with '`#if` *A*' statements in which *A* represents a feature. At build-time, the source code is first configured, a variant describing a particular product is derived by selecting a set of features relevant for it, and only then is this variant compiled or interpreted.

In this two-stage process the compiler handles only the second stage artifacts—the code of the actual product variant. Consequently, all its static analysis mechanisms (such as type checking, data and control-flow analyses) do not analyze the entire program source code, but only the variant specialized for a particular product. This is entirely unacceptable for analyses that aim at identifying program errors. Often, it is not feasible for the vendor shipping the code to analyze each of the variants separately, due to a combinatorial explosion of the number of products (variants). For example, if variability is used to provide personalization of software for various users, it suffices to have 33 independent features to yield more configurations than people on the planet (2^{33}). As little as 320 optional features yield more configurations than the number of atoms in the universe. Now, we have the Linux kernel code base with more than 11,000 features [4]. The problem is particularly burning when run-time errors remain disguised because exhaustive analysis is not possible [5].

In the last decade, many existing program analysis and verification techniques have been *lifted* to work on program families leading to the emergence of so-called *family-based* or *variability-aware analyses* [6]. The main advantage of these analyses is that they do *not* work in two stages, i.e. they do not generate and analyze individual variants separately, but directly analyze the entire code base—all configuration variants at once—at a cost much lower than the accumulated cost of analyzing each of the product variants separately.

Unfortunately, along with the growth of the collection of available lifted analysis methods, a more fundamental worry became increasingly clear: does the variability challenge require redevelopment of the entire language and compiler engineering theory? In response, the industry initiated standardization efforts to codify common understanding of what variability in languages is (for example [7]). In research, a number of papers have started to appear that tackle the more fundamental question of "what is variability in a programming language?" [8]. As part of this larger effort, we attack the problem by developing a systematic understanding of (1) how a single-program analysis relates to the lifted family-based analysis, (2) how programming language definitions (including semantics) are enriched with variability and (3) how a program analysis developed formally for a single program can be systematically lifted into a correct analysis for a family of programs.

We develop a systematic methodology for lifting single-program analyses using abstract interpretation [9]. Abstract interpretation is a unifying theory of sound abstraction and approximation of structures; a well established general framework, which can express many analyses (including data-flow analyses [9], control-flow analyses [10], model checking [11,12], and type checking [13]). Our method exploits knowledge about a single-program analysis to obtain a family-based analysis. The family-based analyses derived using this method are not only sound, but also formally and intimately related to their single program origins. The method is applicable to any analysis expressible as an abstract interpretation, but our focus here is on the constant propagation analysis. The following contributions are made:

(C1) A systematic method for compositional derivation of family-based analyses based on abstract interpretation.
(C2) The correctness (soundness) of the obtained family-based analyses follows by construction.
(C3) Understanding of the structure of the space of family-based analyses (how single-program analyses induce family-based analyses, and which of their abstraction components can be reused at the family level).
(C4) Understanding of individual family-based analyses (in particular, precisely where analysis precision is lost).
(C5) Transfer of the usual benefits of abstract interpretation to family-based analyses (for example, techniques for trading precision for speed and methods for proving analyses to be semantically sound).
(C6) A step-by-step example-driven demonstration of how to derive a family-based analysis.

This work represents an extended and revised version of [14]. Compared to the earlier work, we provide formal and carefully explained proofs of all theorems. We use a running example throughout the paper in order to clarify and improve the presentation of the proposed method and the introduced concepts. In addition, we discuss on an efficient implementation of this method and support our claims by some practical results.

The work is organized as follows. First, a simple imperative language and its operational semantics are presented in Section 2. Then in Section 3, we present a systematic derivation of constant propagation analysis for this language, which is based on the calculational approach to abstract interpretation [15]. In Section 4, we show how the entire derivation process and result can be lifted to the family level for analyzing Software Product Lines. An alternative way to derive lifted analyses of family programs is described in Section 5. We then discuss how the proposed lifted analyses can be efficiently implemented in Section 6. In the end, we discuss related work, and conclude by presenting some ideas for future work.

2. A programming language

We begin by defining the programming language that we want to analyze. Then, we present its operational semantics as we aim to develop a provably sound analysis. Finally, we introduce static variability into the language, and into its formal semantics.

Download English Version:

https://daneshyari.com/en/article/433230

Download Persian Version:

https://daneshyari.com/article/433230

Daneshyari.com