Science of Computer Programming 113 (2015) 191–220

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Designing a verifying compiler: Lessons learned from developing Whiley

David J. Pearce *, Lindsay Groves

School of Engineering and Computer Science, Victoria University of Wellington, New Zealand

ARTICLE INFO

Article history:
Received 30 April 2014
Received in revised form 27 September 2015
Accepted 29 September 2015
Available online 21 October 2015

Keywords:
Program verification
Loop invariants
Hoare logic
Verification tools

ABSTRACT

An ongoing challenge for computer science is the development of a tool which automatically verifies programs meet their specifications, and are free from runtime errors such as divide-by-zero, array out-of-bounds and null dereferences. Several impressive systems have been developed to this end, such as ESC/Java and Spec#, which build on existing programming languages (e.g., Java, C#). We have been developing a programming language from scratch to simplify verification, called Whiley, and an accompanying verifying compiler. In this paper, we present a technical overview of the verifying compiler and document the numerous design decisions made. Indeed, many of our decisions reflect those of similar tools. However, they have often been ignored in the literature and/or spread thinly throughout. In doing this, we hope to provide a useful resource for those building verifying compilers.

1. Introduction

The idea of verifying that a program meets a given specification for all possible inputs has been studied for a long time. Hoare's Verifying Compiler Grand Challenge was an attempt to spur new efforts in this area to develop practical tools [1]. According to Hoare's vision, a verifying compiler "*uses automated mathematical and logical reasoning to check the correctness of the programs that it compiles*" [1]. Hoare's intention was that verifying compilers should fit into the existing development tool chain, "*to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components*". For example, commonly occurring errors could be automatically eliminated, such as: *division-by-zero*, *integer overflow*, *buffer overruns* and *null dereferences*.

The first systems that could be reasonably considered as verifying compilers were developed some time ago, and include that of King [2], Deutsch [3], the Gypsy Verification Environment [4] and the Stanford Pascal Verifier [5]. Following on from these was the Extended Static Checker for Modula-3 [6]. Later, this became the Extended Static Checker for Java (ESC/Java) — a widely acclaimed and influential work in this area [7]. Building on this success was the Java Modelling Language (and its associated tooling) which provided a standard notation for specifying functions in Java [8,9]. Since then a variety of other tools have blossomed in this space, including Spec# [10,11], Dafny [12,13], Why3 [14] and VeriFast [15,16].

Much of the research in verifying compilers has targeted established languages (e.g., Java, C, C#, etc.). Unfortunately, such languages were not designed for verification and contain numerous problematic features, including: fixed-width number representations [17,18], unrestricted pointers [19], arbitrary side-effects [20], closures [21] and flexible threading mod-

* Corresponding author.
E-mail addresses: djp@ecs.vuw.ac.nz (D.J. Pearce), lindsay@ecs.vuw.ac.nz (L. Groves).

http://dx.doi.org/10.1016/j.scico.2015.09.006
0167-6423/

els [22,23]. The alternative, of course, is to design programming languages from scratch specifically for this purpose. Barnett et al. argue this "*lets a language designer pick features that mesh well with verification*" [10].

In this vein, we have developed a programming language from scratch, called Whiley, and an accompanying verifying compiler [24–28]. Whiley is an imperative language designed primarily to simplify verification. The project's goals are:

1. **Ease of Use.** An important goal is to develop a system which is as accessible as possible, and which one could imagine being used in a day-to-day setting. To that end, the language was designed to superficially resemble modern imperative languages (e.g., Python). To simplify verification, Whiley employs unbounded integer and rationals rather than fixed-width arithmetic. Likewise, to further reduce programmer burden, simple loop invariants are inferred where possible.
2. **Scope.** Another important goal of the project is to enable programs to be automatically verified as: *correct with respect to their declared specifications*; and, *free from runtime error* (e.g., divide-by-zero, array index-out-of-bounds, etc.). However, more complex properties such as *termination*, *worst-case execution time*, *worst-case stack depth*, etc. are not considered (although would be interesting future work). This impacts the language design as, for example, we do not provide syntax for expressing loop variants (i.e., establishing termination is not a consideration).
3. **Demonstration.** Another important goal of the project is to demonstrate that Whiley is suitable for developing safety-critical systems. This means, for example, that Whiley programs must be executable and can be integrated into existing systems (i.e., via a foreign function interface). To this end, we are initially targeting small embedded systems (see Section 8).

Many of these goals are seemingly at odds with each other. For example, unbound arithmetic is not suitable for embedded systems. In many cases, we can work around this by exploiting function specifications as these provide rich information [29]. When compiling for an embedded device we can, for example, require all integers used within a function are bounded using appropriate invariants (i.e., that they are required to be within certain ranges, etc.). Thus, the difficulty of verification is layered. That is, users need not initially worry about bounded arithmetic and, instead, can focus on learning and understanding the verification process. Later on, if they wish to target an embedded system, they must then further refine their specifications to allow this. To these ends, we have successfully compiled Whiley programs to run on a quadcopter (see Section 8). Likewise, we have used Whiley to teach students at Victoria University of Wellington about verification (see Section 8). Finally, the Whiley verifying compiler is released under an open source license (BSD), and can be downloaded from http://whiley.org and forked at http://github.com/Whiley/. All examples in this paper have been tested against the latest release at the time of writing (version 0.3.36).

1.1. Contribution

The primary contribution of this paper lies in documenting numerous important issues faced in developing a verifying compiler and the decisions we made. Many of our choices reflect those of similar systems such as Dafny, Spec#, ESC/Java, etc. Such decisions are often left undocumented, or spread throughout numerous papers in the literature. By bringing these together in one place we hope to provide a useful resource for those designing and implementing verifying compilers. Note, however, this paper does not attempt to evaluate our language design against those goals outlined above, and this remains as future work.

Finally, an earlier version of this paper was published at the Workshop on Formal Techniques for Safety-Critical Systems (FTSCS'13) [30]. We present here a significantly revised and extended version of that paper, which includes additional discussion of our experiences, a small evaluation against benchmarks from the COST'11 [31] and VSCOMP'10 [32] verification competitions and three case studies looking at Whiley in the context of teaching, embedded systems and alternative SMT solvers.

1.2. Organisation

We provide a general introduction to verification with Whiley in Section 2, followed by an overview of the compiler architecture in Section 3. In Section 4, we provide a detailed discussion of the salient design choices made and their implications in practice. Then, in Section 5 and Section 6 we reflect on experiences gained from verifying programs with Whiley. Following this is a small evaluation of Whiley's verification capability in Section 7 and a discussion of three case studies in Section 8. Finally, related work is discussed in Section 9 before we conclude in Section 10.

2. Language overview

In this section, we introduce the Whiley language in the context of software verification through a series of examples. However, we do not provide an exhaustive examination of the language and, instead, the interested reader may find more detailed introductions elsewhere [33,34].