How Tests and Proofs Impede One Another: The Need for Always-On Static and Dynamic Feedback

Michael D. Ernst

Computer Science & Engineering
University of Washington
Seattle, WA 98195-2350, USA
mernst@cs.washington.edu

Abstract. When creating software, developers rely on feedback in the form of both tests and proofs. The tests provide dynamic feedback from executing the software. The proofs provide static, conservative feedback from analyzing the source code. Testing is widely adopted in practice, but type systems are the only variety of proof to achieve widespread adoption.

Dynamic and static feedback provide complementary benefits, and neither one dominates the other. Sometimes, sound global static checking is most useful. At other times, running tests is most useful. Unfortunately, current languages and IDEs impose too rigid a model of the development process. As a result, the developer is not in control of the development process. In situations where a small amount of appropriate feedback could have yielded insight, the developer must choose between either static or dynamic feedback, and warp his or her development style to the limitations of the language. This situation is wrong: the developer should always have access to immediate execution feedback, and should always have access to sound static feedback.

For example, in a typical statically-typed language, a developer is prevented from running the program if any type errors exist, even if they are not germane to the particular test. In a dynamically-typed (or weakly-typed) language, a developer is free to build insight by experimenting, but never gets the assurance of a proof of type correctness.

We need a better approach. A programmer should be able to view and execute a program through the lens of sound static typing. If the compiler issues no type errors, that is a guarantee (a proof) of static type soundness. A programmer should also be able to view and execute the same program through the lens of dynamic typing with no statically-imposed restrictions. The run-time system should suppress static type errors, unless they lead to user-visible failures, or the developer wishes to examine them. Furthermore, the programmer should be able to switch between these two views as often as desired, or to use them both simultaneously.

A typical use case is: a developer starts from an existing statically-typed codebase (or start writing new code), uses both dynamic and static feedback, and finally restores static type-correctness. Because the developer has types in mind and even uses the type system, large variations from statically-typeable idioms are unlikely.

G. Fraser and A. Gargantini (Eds.): TAP 2010, LNCS 6143, pp. 1–2, 2010.
© Springer-Verlag Berlin Heidelberg 2010

The ability to execute and run code at any moment is useful in many circumstances, including software creation (e.g., prototyping) and software evolution (e.g., representation or interface changes, library replacement, exploratory changes). Approaches such as prototyping in a dynamic language then rewriting with types, or viewing one part of the program as typed and another part as untyped, address only a subset of these scenarios and impose a significant burden on the developer.

I will describe additional background, theory, and practical details. I will also share experience with an implementation, Ductile, that supports the goal of always-on static and dynamic feedback.