Position Paper: Performance Evaluation for Gradual Typing

Asumu Takikawa Daniel Feltey Ben Greenman Max S. New
Jan Vitek Matthias Felleisen
PRL, Northeastern University
{asumu, dfeltey, types, maxsnew, j.vitek, matthias}@ccs.neu.edu

Abstract

Gradually typed programming languages aim to improve software maintenance by allowing programmers to selectively add type annotations to untyped programs. Run-time checks ensure that these typed portions interact soundly with unannotated parts of the program. These checks, however, may introduce unacceptable performance overhead. The extent of the overhead has not been systematically studied and no common methodology exists to diagnose such problems. In this position paper, we propose an idea for a framework for evaluating the performance of a gradual type system.

1. The Gradual Typing Promise

Gradually typed programming languages promise to improve software maintenance for untyped scripting languages. Using such systems, programmers may selectively add type annotations to their existing untyped programs. The annotated parts are checked, and run-time contracts or casts ensure that they safely interact with the remaining untyped portions.

Programmers use gradual type systems in order to realize software engineering benefits from types such as enforcing documentation, guiding refactoring, and catching regressions. In addition, the gradual typing promise implies that as programmers add type annotations, their program will continue to run. This part of the promise is held up by allowing typed and untyped code to link together with inserted run-time checks. For a gradual type system to be a net benefit, it should also allow gradually typed programs to remain *performant* as they are incrementally converted. Therefore, it is desirable for a gradual type system to promise low overhead for interoperation.

In our experience, existing gradual type systems (including the systems we maintain) fail to meet this criterion. Gradual type systems in the literature report slowdowns of 72x (Rastogi et al. 2015), 10x (Vitousek et al. 2014), and 4x (Takikawa et al. 2015) in programs due to the insertion of dynamic checks. Practical users of gradual type systems have also reported 25-50x slowdowns.[1]

To make gradual type systems live up to their promises, we must (1) diagnose what kinds of programs and what degree of "typedness" leads to performance problems, and (2) identify the tools, language features, or implementation techniques that will help eliminate the overhead. This position paper focuses on the diagnostic side, and outlines some potential solutions.

2. The State of Gradual Type System Evaluation

Despite a wealth of literature on gradual typing, there is a dire lack of performance evaluation efforts. As mentioned, several projects have reported slowdowns on example programs, and others have explored the cost of the checking mechanism itself (Allende et al. 2013b) but these results are difficult to compare and interpret in the broader context of the software engineering benefits that gradual type systems promise.

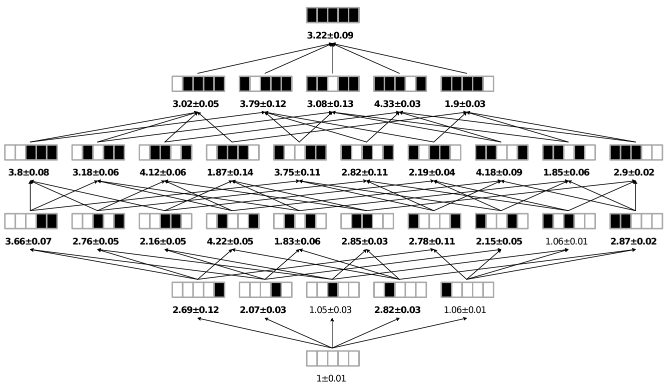

Figure 1: Lattice example with five modules

In part, this points to a lack of any accepted methodologies for evaluating gradual type system performance. Such a methodology should provide a systematic approach to evaluating interoperation overhead. Here, we propose steps towards the development of an evaluation setup that tries to discover the potential overhead.

3. Exploring the Program Space

To work towards a methodology, we need to first understand how gradual type systems are used. The basic premise is that programmers do not add type annotations to an entire program at once. Instead, programmers can choose intermediate states in which some parts of the program are typed and others are untyped. The granularity of the type-checked parts—by module, by block, or by expression—depends on the gradual type system.

For our evaluation, we focus on Typed Racket—a gradually typed sister language to Racket—because of its maturity as a gradual type system; it has been in development since 2006. Typed Racket is a *macro*-level gradual type system, which means that types are added to the program at module granularity and dynamic checks are installed at these boundaries between typed and untyped modules. As a result, Typed Racket does not need to instrument untyped modules at all, which enables separate compilation within gradually typed programs.

This approach is in stark contrast with the *micro*-level approach, in which typed and untyped code is mixed freely in a single module. Variables without type annotations are assigned the type `Dyn`.

[1] `http://docs.racket-lang.org/math/array.html`

These variables induce casts when typed portions of the program expect more specific types.

Recognizing that programmers gradually add types to their program, we propose to look at *all possible ways in which a programmer could add types to a program* in the context of the macro approach. Specifically, we take existing Racket programs, come up with type annotations for all of the modules in the program, and then consider the possible typed/untyped configurations of modules. We then *benchmark all of these possible configurations* to determine the performance overhead of run-time checks by comparing against the original program.

Given n modules in the program, this produces 2^n configurations of the program. We can represent this space of configurations as a lattice in which the nodes represent a particular configuration of modules in a program—that is, whether each module is typed or untyped. An edge between two nodes A and B indicates that configuration A can be turned into configuration B by adding type annotations to a single additional module. See figure 1 for an example of a program lattice (for a program that traverses Racket bytecode data structures). The bottom of the lattice represents the original, fully untyped program and the top of the lattice represents the program with types added to all modules.

The labels on the nodes represent the normalized runtimes (mean and standard deviation) of benchmarks that we run on the whole program. The black and white boxes represent whether a module is typed (black) or untyped (white). Note that since a program may call out to additional libraries, the top of the lattice (the fully typed program) may still have run-time overhead.

Paths in the graph that start from the bottom correspond to the timeline of a hypothetical programmer who is adding types to the program. Ideally, most configurations of the program should have reasonable overhead. In practice, however, large portions of the lattice will contain regions of poor performance due to, for example, tightly coupled modules with dynamic checks on the boundary. Based on these lattices, we hope to understand to what degree these regions of poor performance affect programs and what kinds of typed-untyped boundaries are especially problematic.

As a first attempt, Takikawa et al. (2015)—including several of the present authors—worked on a small-scale version of this approach in the context of Typed Racket. Following up, we are working on scaling this evaluation idea to programs with a larger number of modules (and hence a much larger number of variations) and are investigating both functional and object-oriented programs.

4. Request for Comments: Scaling the Idea

The large number of variations makes data visualization and analysis challenging. We are therefore considering alternatives to the lattice form of visualization such as histograms over path metrics and heatmaps.

Although our idea is straightforward for the macro style of gradual typing, it is not obvious how to apply it to the micro approach that is common in other systems such as Gradualtalk (Allende et al. 2013a), Reticulated Python, and Safe TypeScript. Specifically, it is not clear how to set up the space of variations. For example, type annotations could be toggled by function, by module, or even by binding site. Picking the latter would lead to a particularly large configuration space since every variable multiplies the number of variations by two.

5. Investigating Potential Solutions

After diagnosing the kinds of overhead found in gradually-typed programs, we intend to investigate possible solutions. Solutions may come in the form of mitigation, in which a tool or language feature helps avoid problematic dynamic checks. Alternatively, the solutions may instead seek to reduce the cost of the checks.

One form of mitigation we have identified is to guide the programmer to good paths through the state space using techniques such as Feature-specific Profiling (St-Amour et al. 2015) with contracts/casts as the target feature.

We also intend to investigate the use of tracing JIT compilation based on the Pycket work by Bolz et al. (2014). The Pycket authors report dramatic reductions in contract checking overhead in untyped Racket programs (Bauman et al. 2015). We are interested in seeing if tracing also benefits the kinds of contract usages that we see in gradually typed programs.

6. Conclusion

Runtime overhead for gradually-typed programs is a pressing concern. Industrial groups[2] continue to adopt unsound interoperation citing performance concerns with run-time checks. However, there are open questions in both diagnosing where these overheads occur and in solving them. Here we propose a framework for diagnosing such overheads by visualizing the effect of adding types to existing programs on runtime performance along various gradual typing paths. Using the diagnostic information, we hope to drive efforts in both tooling and compilation for gradually typed languages.

Bibliography

E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013a.

E. Allende, J. Fabry, and É. Tanter. Cast Insertion Strategies for Gradually-Typed Objects. In *Proc. DLS*, 2013b.

S. Bauman, C. F. Bolz, R. Hirschfield, V. Kirilichev, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Pycket: A Tracing JIT For a Functional Language. In submission, 2015.

C. F. Bolz, T. Pape, J. G. Siek, and S. Tobin-Hochstadt. Meta-tracing makes a fast Racket. In *Proc. DYLA*, 2014.

A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *Proc. POPL*, 2015.

V. St-Amour, L. Andersen, and M. Felleisen. Feature-specific Profiling. In *Proc. CC*, 2015.

A. Takikawa, D. Feltey, E. Dean, R. B. Findler, M. Flatt, S. Tobin-Hochstadt, and M. Felleisen. Towards Practical Gradual Typing. In *Proc. ECOOP*, 2015.

M. M. Vitousek, A. Kent, J. G. Siek, and J. Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. DLS*, 2014.

[2] For example, Hack for PHP and Flow for JavaScript.