Optional Types

Facebook created and continues to develop two optionally-typed languages: Hack and Flow. These languages reconcile existing and useful untyped code with the engineering benefits of optional, machine-checked type annotations.

Optional type systems are a worthy investment over untyped code because they improve developer productivity along several dimensions including development speed, maintainability, testing, and debugging. Development speed increases by quick detection of logical errors and IDE tools like type-enabled autocompletion. Maintenance comes from the form of documentation provided by type annotations—programmers need not guess or read comments to learn how to invoke a typed function. Testing becomes easier in typed languages because some faulty programs now fail to compile; for example, both Hack and Flow statically detect many null pointer exceptions in typed code. Similarly, bugs are easier to find when developers have type information to mediate their design with the program's runtime behavior.

Still, the benefits of optional type systems pale in comparison to those realized by sound type systems. In a sound language, for example OCaml or Haskell, the type system *eliminates* a class of bugs and enables formal reasoning about program correctness. In contrast, Hack & Flow make no guarantees about the behavior of typed functions called with untyped arguments or untyped functions called from typed code—both of which are perfectly legal operations. This makes it both incorrect and *unsafe* to think of optional types as anything more than a linting tool. Erroneous or malicious untyped scripts can break the type system's assumptions at any point during execution. When this happens, the program will *continue running*—potentially leaking secure information or corrupting data—unless stopped by a chance assertion or segfault.

Admittedly, these same problems arise in so-called sound typed languages when using an FFI or marshalling values. Interaction between typed and untyped code is dangerous! Facebook's decision to support full interoperability with their existing untyped product only makes the issue unavoidable—we might sidestep the problem in Haskell by informally verifying the untyped runtime system and calling it "trusted", but this strategy does not scale. What we need is a *gradually typed* programming language: a language that provides type-*sound* interaction between typed and untyped code.

Sound Gradual Typing

For the past decade, my advisor's research group has built and maintained a gradually typed language [3]. This language's type system supports sound interaction with untyped code even in the presence of advanced features such as flow-guided type refinement, variable-arity polymorphism, first-class classes, delimited continuations, and hygenic macros.[1] All type errors originating in typed modules are caught statically, and any runtime type errors are detected immediately and attributed to the boundary between typed and untyped code that violated the type system's invariants.

Soundness is enforced by compiler-generated assertions that check every runtime use of an unsafe boundary. These assertions provide strong safety guarantees and might be the key to offering type soundness in Hack and Flow, but the current implementation carries a high performance cost. Quite often, this cost is enough to render a program too slow for practical use, forcing the developer to either try adding more type annotations or revert to the working untyped code. Achieving sound and performant gradual typing is an open problem and the subject of my research.

Agenda

When I joined Northeastern last Fall, I began research on gradual typing in two directions. The first is improving the current technology by addressing performance bottlenecks. The second is a fundamentally new implementation strategy, where the type system is built from modular and extensible components.

Performance Evaluation

With collaborators at Northeastern, I developed an evaluation framework for gradual type systems. Using this framework, I evaluated a dozen moderately-sized programs.[2] For a given program with N modules,

[1] There are a few limits on these features' use across type boundaries e.g., typed macros cannot be used in untyped code.
[2] The largest was 6,700 lines of code across 16 modules.

we tested the 2^N programs obtained by making each module typed or untyped. For each of these 2^N programs, we measured its performance overhead (relative to the untyped program) and distance from a configuration with acceptable performance. This work will appear at the next POPL [2].

Our results were appalling. Although the fully-typed versions of programs were often slightly faster (~10%) than the untyped ones, gradual typing frequently introduced order-of-magnitude slowdowns. There was seldom even a migration path from fully-untyped to fully-typed that avoided these extreme performance costs.

Examining the details of each benchmark, we found that a large proportion of the overhead was due to flat contracts protecting data structures. Typed users of untyped structures naturally had to pay for each API call, but so did untyped users of typed structures—even when the untyped code did not inspect or mutate the typed data between interactions. Hence any operation on a typed data structure invoked by untyped code, no matter how trivial, imposed an $O(n)$ cost when called on a structure of n elements.

I am investigating techniques for reducing these costs. One idea is to develop software contracts that are aware of their type and can skip redundant checks on values. Another is to exploit the abstraction boundaries implicit in programs so that developers can see the performance gains of opaque datatypes without globally enforcing opaque usage. As I explore these and other ideas, I am also building tools to automate the process of analyzing performance and recommending a course of action. My work so far has led to one tool for tracking the run-time flow of values in a program,[3] and I am currently building an interactive type inference engine.

Reusable Type Systems

There is a gap between type systems' description and implementation. On paper, we formulate type systems as a composable set of rules or derivations. Taking a few of these rules as the starting point for a new type system is easy and encouraged. In code, a type system is part of the language's compiler. Changes to the type system are effectively changes to the language. One does not simply modify the type checker without forking the language or earning the core maintainers' approval.

Dr. Stephen Chang, one of my advisor's post-docs, noticed this distinction and invented a type system design where new types and judgments are as easy to add as new functions or libraries. His vision is a growable type system—just as libraries of functions aid programming, these composable and extensible type systems promote application-specific reasoning instead of forcing all developers into a single logic.

Under Dr. Chang's guidance, I have implemented a variety of type systems in this style and explored their interactions. The results have been encouraging. For example, I implemented rules for occurrence typing [4] and parametric overloading [1] that independently build on the simply-typed lambda calculus. In a separate extension, I then combined these rules by writing an algorithm to resolve ambiguity where possible and otherwise use runtime type dispatch. Our other type systems follow this same approach of incremental construction and certification in user-facing libraries.

We are currently exploring gradually-typed languages in this style. Besides being simple to extend, the type rules for these languages control their elaboration into source syntax. This makes typed syntax fully compatible with untyped code and allows context-sensitive elaboration strategies, thereby enabling aggressive type-driven optimizations.

Applicability

The results of my work will suggest possibilities for Facebook to implement type soundness without sacrificing runtime efficiency or adding the unrealistic assumption that 100% of the code used by any Hack or Flow application has been approved by the type checker. Type soudess will make code easier to maintain, test, and debug because it guarantees the currently-informal semantic properties of types and ensures protection from the wide class of bugs known as "type errors".

[1] Stefan Kaes. Parametric overloading in polymorphic programming languages. In *ESOP*, 1988.

[2] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? To appear in *POPL* 2016.

[3] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *DLS*, 2006.

[4] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP*, 2010.

[3] https://github.com/bennn/tracers/tree/master/count-predicates