Research Statement

Ben Greenman
2021-11-29

As software systems grow to meet a variety of demands, programming languages should support a variety of tools to facilitate the construction of multi-component software systems. Two useful tools are static and dynamic type systems, both of which enable abstraction-safe program components. Other tools include SAT solvers, proof assistants, and domain-specific languages. Developers often need to combine tools; for instance, by writing typed code that interacts with a solver. But unless the tool designers specifically planned for all possible interactions, there is a risk of unsound compositions in which one component misinterprets the data sent by another. These soundness holes threaten correctness and security.

My research agenda is to develop methods for reasoning about multi-component systems, specifically their **formal guarantees**, **performance implications**, and **ergonomics**. In short, I build methods for three essential P's: **proofs**, **performance**, and **people**.

A promising way forward is to identify useful combinations of tools and study their interactions in depth. In this spirit, research on gradual typing has investigated combinations of static and dynamic type systems. Static types impose compile-time constraints to predict the behavior of code. Dynamic types insert run-time checks to form a safety net. Combining both has significantly enriched our understanding of the guarantees that compile-time checks can provide and of run-time methods for enforcing these guarantees. I believe that systematic work on other combinations will improve the reliability of software and open further research opportunities.

Dissertation Work

Over the past two decades, gradual typing has emerged as a promising solution to the dilemma between typed and untyped (or, dynamically typed) programming languages. Researchers in the area have created type systems that accommodate untyped code and runtime systems that monitor the interactions among components. These ideas seemed to offer the best of two worlds and generated a great deal of interest—so much that industry teams at Google, Facebook, Microsoft, Adobe and Dropbox built their own gradual languages. Today, there are at least twenty such languages available to programmers.

Gradual typing does not yet, however, live up to the "best of both worlds" promise. Each gradual language represents a point in a complex design space where static type guarantees imply run-time costs. Understanding and reducing these costs is a major challenge. The large number of languages demonstrates that many strategies exist, but these points say little about how the strategies relate to one another. In short, the area is lively but disorganized.

My research adds order to the gradual design space. I have built methods that enable scientific comparisons of gradual languages along two key dimensions: run-time performance and formal guarantees. These methods have elevated the state of affairs from vague claims about different languages [8] to formal arguments and rigorous data [7, 9, 11, 12].

- *Performance [10, 14].* Gradual type systems allow any mix of typed and untyped code to run, but make no guarantee about run-time overhead. In fact, performance can be arbitrarily bad. To understand whether such costs are widespread and to motivate language improvements, I designed the first **comprehensive and scalable method to measure gradual typing performance**, curated two benchmark suites [1, 8], and measured several versions of two gradual typing systems [4, 7, 11].

- *Formal Guarantees [2, 9].* The strategy that a gradual language uses to enforce types can change the behavior of programs. A firm understanding of these changes is essential, but difficult to arrive at because two languages may interpret similar-looking types in different ways. I therefore developed the first **formal account of type-enforcement strategies** using a model that provides a common ground and a tower of theorems to clarify their behavioral implications. The theorems intentionally do not set criteria for what a "best" strategy ought to do. Rather, they give positive characterizations of points in the design space.

At present, no single gradual language provides both strong guarantees and fast performance without compromising expressiveness. Researchers have explored two approaches to close these gaps: building a new language and building a new compiler. Neither is ideal because they impose a migration cost on programmers. It would be better if an existing language or compiler could be adapted. To this end, I extended a gradual language with strong type guarantees to incorporate a semantics that explores a complementary point in the design space [5, 6]. The result is the **first language to support interoperability between two sound gradual semantics**. Performance bottlenecks are rare in this language; switching between type enforcement strategies on a whole-sale basis addresses most pathologies. Fine-grained combinations can further improve performance and let programmers deploy critical type guarantees as needed.

Ongoing Work

Despite our recent success in evaluating *proofs* and *performance*, the gradual typing community has largely neglected the third crucial "*p*" of good programming languages research: *people.* I have conducted a first study in this direction [15] and the community's positive response indicates a willingness to heed the results of further research. My ongoing work as a **CIFellows 2020** postdoc at Brown is therefore **focused on human factors**. In particular, I am collaborating with a team at Instagram to assess a language they have created which imposes a coding discipline but in return achieves both soundness and performance. The formalization has already found several bugs, including one that led to a segfault.

This postdoc appointment has also been an opportunity to broaden the scope of my research. In addition to gradual typing, I have studied misconceptions regarding Linear Temporal Logic (LTL) [3] and type systems for tabular data [13]. The LTL work is a key step toward tools that assist developers in the construction of secure and responsive systems. Types for tables are needed to bring the reliability of static types to data science applications.

In the future, my goal is to leverage these experiences in human-factors research and address all 3 P's of language design questions (proofs, performance, and people) throughout my career—whether in gradual typing or in other research topics.

Future Work

Within a few years, I predict that every modern language will be gradually typed. Dynamic languages will add at least a syntax for type annotations and maintenance tools that leverage the types. Python and JavaScript have already done so with unsound types. Static languages will follow C# and add a dynamic type to express objects that originated in untyped components. Researchers can help language designers by mapping the design space and recommending useful points. As practictioners apply gradual typing to a full-fledged language and discover challenges, researchers can also explore solutions. Thus, gradual typing will become increasingly important along these two research vectors: developing semantics and supporting practical aspects of existing languages.

Other important future directions lie beyond the scope of typed/untyped combinations. Techniques for sound and efficient gradual types have implications for other multi-component systems. These include both sibling-language systems that resemble the typed/untyped sitation and others that combine totally different languages. Concrete topics include the following:

- *Security.* Multi-component systems are vulnerable to attacks that send unexpected data across a boundary. A static analysis, e.g., via a type system for information flow, can identify potentially-unsafe boundaries and put a conservative bound on the code that an attack can reach. Such an analysis must be aware that attacks can originate from un-analyzed components.

- *Lifetime Analysis.* The Rust programming language has shown that a static lifetime analysis can detect aliasing bugs and create opportunities for concurrent execution. Other systems languages would benefit from a similar analysis so that programmers need not port their application to Rust to prevent data races. The analysis may target a restrictive subset of the language as long as it can recognize unsafe boundaries to legacy components.

- *Probabilistic Extensions.* Many probabilistic programming languages extend a general-purpose host language with an API to a probabilistic model. The model can be used to solve machine learning problems, but its interactions with the host language are error-prone. A gradual type system could ensure that the host language sends sensible data to the model.

- *Solver-Aided Programming.* The Alloy modeling language lets programmers specify aspects of a software system and uses a solver to find bugs in the specification. Because the solver works at a lower level of abstraction than the modeling language, users have to remember that it may not respect "obvious" system invariants. Solver-aided languages would benefit from a tool-assisted method for conveying invariants to the solver, or for detecting when an important property has been violated.

Toward these goals, gradual typing contributes lessons about how to identify critical boundaries, how to protect the data that crosses these boundaries, and how to efficiently build the protection layer. My expertise with proofs, performance, and people will inform methods that address the challenges.

[1] GTP Benchmarks. URL `https://docs.racket-lang.org/gtp-benchmarks`. Accessed 2021-11-10.

[2] Anonymous Author(s). How to evaluate the semantics of gradual types. *Submitted for publication*, 2021.

[3] Anonymous Author(s). Little tricky logic: Misconceptions in the understanding of LTL. *Submitted for publication*, 2021.

[4] Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA):133:1–133:27, 2018.

[5] Ben Greenman. *Deep and Shallow Types*. PhD thesis, Northeastern University, 2020.

[6] Ben Greenman. Deep and shallow types for gradual languages. *Submitted for publication*, 2021.

[7] Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *PACMPL*, 2(ICFP):71:1–71:32, 2018.

[8] Ben Greenman and Zeina Migeed. On the cost of type-tag soundness. In *PEPM*, pages 30–39, 2018.

[9] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. Complete monitors for gradual types. *PACMPL*, 3(OOPSLA): 122:1–122:29, 2019.

[10] Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. How to evaluate the performance of gradual type systems. *JFP*, 29(e4):1–45, 2019.

[11] Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. A transient semantics for Typed Racket. *Programming*, 6(2):1–26, 2022.

[12] Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. How to evaluate blame for gradual types. *PACMPL*, 5(ICFP):68:1–68:29, 2021.

[13] Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi. Types for tables: A language design benchmark. *Programming*, 6(2):1–30, 2022.

[14] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *POPL*, pages 456–468, 2016.

[15] Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, and Shriram Krishnamurthi. The behavior of gradual types: a user study. In *DLS*, pages 1–12, 2018.