Benjamin Lee Greenman
Brown University
CIT building
115 Waterman St
Providence RI 02912
`benjamin.l.greenman@gmail.com`
`781-924-9989`

Research Interests

General interests: Language design issues regarding proofs, performance, and people. What guarantees can a language offer, how efficiently can it run, and to what extent does it help users meet their goals?

Specific interests: Migratory Typing, Language Interoperability, Type Theory, Formal Methods

Education

- Northeastern University — 2014 – 2020

Degree	Ph.D
Area	Programming Languages
Advisor	Matthias Felleisen
Thesis	Deep and Shallow Types

- Cornell University — 2013 – 2014

Degree	Master of Engineering
Major	Computer Science
Advisor	Ross Tate

- Cornell University — 2010 – 2013

Degree	Bachelor of Science
Major	Industrial and Labor Relations
Minor	Computer Science

- Hudson Valley Community College — 2009 – 2010
  General Studies

Employment

- Brown University — 2021 – ongoing
  Postdoctoral Researcher, CIFellows 2020
- Knightsbridge Park — 2017
  Consultant, Web Scraping
- Cornell University — 2012 – 2014
  Research Assistant

- Rentenna Inc. — 2012 – 2014
  Software Engineering Intern

Teaching

- Topics in PL and Systems: Tables and Humans — 2020
  Organizer
- Software Development — 2018, 2020
  Teaching Assistant
- Fundamentals I (Computing and Programming) — 2016
  Teaching Assistant
- Object-Oriented Design — 2016
  Teaching Assistant
- Functional Programming and Data Structures — 2012 – 2014
  Teaching Assistant

Students Supervised

- Qianfan Chen — 2020 – ongoing
  Sc.B., Brown University
- Kuang-Chen Lu — 2020 – ongoing
  Ph.D., Brown University
- Milo Davis — 2017
  B.S., Northeastern University
- Zeina Migeed — 2016 – 2017
  B.S., Northeastern University

Awards

- CRA/CCC/NSF CI Fellowship — 2021 – ongoing
- SIGPLAN Student Scholarship to: 50 Years of the ACM A.M. Turing Award — 2017
- Northeastern CCIS Graduate Community Service Award — 2016
- Cornell CS Teaching Award — 2014
- Cornell CS Teaching Award — 2013

Professional Service

- Artifact Evaluation Committee Co-Chair — OOPSLA 2022
- Program Committee — ICFP 2021
- Program Committee — PLDI 2021
- Artifact Evaluation Committee — ECOOP 2017
- Artifact Evaluation Committee — OOPSLA 2017
- Artifact Evaluation Committee — OOPSLA 2016

Publications

Journal

- Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. — JFP 2019
  How to Evaluate the Performance of Gradual Type Systems

Conference, Symposium, and Hybrid Conference / Journal

- Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen — Programming 2022
  A Transient Semantics for Typed Racket
- Kuang-Chen Lu, Ben Greenman, and Shriram Krishnamurthi — Programming 2022
  Types for Tables: A Language Design Benchmark
- Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas — ICFP 2021
  How to Evaluate Blame for Gradual Types
- Ben Greenman, Matthias Felleisen, and Christos Dimoulas — OOPSLA 2019
  Complete Monitors for Gradual Types
- Preston Tunnell Wilson, Ben Greenman, Justin Pombrio, Shriram Krishnamurthi. — DLS 2018
  The Behavior of Gradual Types: A User Study
- Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St. Amour. — OOPSLA 2018
  Collapsible Contracts: Fixing a Pathology of Gradual Typing
- Ben Greenman, Matthias Felleisen. — ICFP 2018
  A Spectrum of Type Soundness and Performance
- Ben Greenman, Zeina Migeed. — PEPM 2018
  On the Cost of Type-Tag Soundness
- Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. — SNAPL 2017
  Migratory Typing: 10 Years Later

- Stephen Chang, Ben Greenman, and Alex Knauth. POPL 2017
  Type Systems as Macros
- Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. POPL 2016
  Is Sound Gradual Typing Dead?
- Ben Greenman, Fabian Muehlboeck, and Ross Tate. PLDI 2014
  Getting F-Bounded Polymorphism into Shape

Workshop

- Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. STOP 2015
  Position Paper: Performance Evaluation for Gradual Typing

Invited Talks

- Racket Con 2020
  Shallow Typed Racket
- Boston University POPV Seminar 2020
  Complete Monitoring for Gradual Types
- GRACE Workshop 2018
  Three Approaches to Gradual Typing

Volunteering

- Bootstrap Professional Development Summer 2021
  Teaching Assistant
- Housing Chair SPLASH 2018
- Northeastern CCIS Hiring Committee Spring 2018
  Student Representative
- PRL Offsite Fall 2019
  Organizer
- Each One Teach One Fall 2015
  AP Java Tutor
- Student Volunteer OOPSLA 2019; Turing Celebration 2017; POPL 2016, 2018; PLDI 2016; ICFP 2015, 2018; ECOOP 2015, 2016
- Ithaca Media Arts Summer 2012
  Teacher, LEGO Mindstorms Camp
- Cornell Math Explorers Winter 2011
  Module Designer

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.

Teaching Statement

Ben Greenman
2021-11-29

Teaching is very important to me. The main reason that I am applying to faculty jobs rather than industry positions is to have the opportunity to teach a variety of students. Why? For one, I enjoy the interactions and challenges that come with teaching. Students always end up teaching me something new, at least by sharing their perspectives, and some make intellectual contributions to research projects. On a deeper level, teaching is meaningful. Computing skills are useful for any person to know and essential for many jobs in todays' workplace.

Experience

Teaching. At Brown, I organized and managed a graduate-level seminar focusing on type systems for data science—specifically, for tabular data. This seminar inspired a student-led research project on type system design. The work is ongoing, but has already produced one publication. At Northeastern, I had four TA opportunities. I was the managing TA in an upper-level course on software development for two semesters. I was a TA for the introductory computing course for one semester, during which I ran weekly lab sessions for approximately 30 students. And I was a TA for a sophomore-level course on object-oriented design for one semester. These experiences shaped my teaching philosophy: programming is best taught by rewarding systematic designs over functional-but-unprincipled solutions. At Cornell, I spent five semesters as TA for a sophomore-level course on functional programming. I gave weekly lectures to ~20 students and helped manage the course in four of these semesters. The department recognized my efforts with a **teaching award** two years in a row. My proudest course-management moment came on the eve of a first exam, when I led a few TAs in rewriting the extremely difficult test that our overloaded professor had drafted. (The rewritten test was still too hard.)

Mentoring. I am currently supervising one Ph.D student and one research undergraduate at Brown. The Ph.D student is seeking a dissertation topic at the intersection of gradual type systems and data science; toward that goal, we have designed a benchmark for table types and are formalizing an industry partner's gradual language. The undergraduate is developing a staged modeling language to teach formal methods. At Northeastern, I supervised two research undergraduates on topics in gradual language implementation. One of these students made significant contributions to a research paper and is currently a programming languages Ph.D student at UCLA. The other went to a blockchain startup after graduating.

These talented students have been a pleasure to work with as I learn to give effective feedback and guidance. My general advising plan is to help students set goals and then give them freedom to accomplish the goals. One practice that I have found useful is to keep a shared diary of long-term ideas and concrete next steps. The long-term notes shape priorities, the next steps ease context switches, the diary format is simple to maintain, and the document itself provides a meeting point outside the student's day-to-day workspace. For remote advising over the past year, these diaries have helped us stay connected despite the lack of shared office space.

Volunteering. I have volunteered with the Bootstrap program while at Brown [4]; with E1T1 [1], an after-school program for underprivileged high schools, while at Northeastern; and with both Ithaca Media Arts [2] and the Math Explorers' Club [3] at Cornell. These experiences gave me a chance to reach a wide variety of learners, ranging from high school teachers to elementary school students. Finding ways to connect with the individuals in these groups was a difficult but exciting challenge. Northeastern recognized my tutoring at E1T1 with a **graduate community service award** [5].

Teaching Goals

At the undergraduate level, CS education desperately needs methods to deliver quality education at scale. Lecture, grading, and feedback need to accommodate hundreds of students at a time. My experiences at Northeastern and Brown have fortunately acquainted me with two promising tools to start from:

- *The Design Recipe*: A structured method for writing programs. The recipe gives students milestones to target, provides structure to Q/A sessions, and lets graders focus on the quality of solutions instead of mere functional correctness.
- *Quizius*: A machine-learning platform that uses student input to discover which concepts the class struggles the most with. Student input lets teachers focus on curating questions (rather than inventing them) and may reveal expert blind spots.

Tools like these, which add structure to classroom discussions and provide automated feedback, are essential to scale and diversify CS classrooms. In particular, students who lack confidence as programmers (and might drop out of an unstructured sink-or-swim course) will likely benefit from clear milestones. As a faculty member, I will use these tools and support the development of others.

At the graduate level, my goal is to supervise a small pipeline of Ph.D students in an apprenticeship style. Each student must work individually first of all and carve out a distinct research topic, though a junior student may briefly pair up with a senior student to learn the ropes. I will provide one-on-one feedback, primarily on research and communication skills.

Courses

I would be eager to teach programming, programming languages, and related topics (such as algorithms, discrete math, theory of computation) at any level. I am especially interested in four such courses:

- an upper-level programming languages course,
- an upper-level software course based on code review and face-to-face communication,
- a course on logic as a tool for software engineers, and
- an intro-level programming course that focuses on systematic design.

Teaching at the introductory level is on this list of interests because it presents an opportunity to stimulate non-major and first-year students. For non-majors, an intro course may transform their careers by imparting critical-thinking skills and programming confidence. For the first-year students, the intro course may spark a research interest that they pursue during their remaining years. I experienced both these effects myself as a first-year non-major at Cornell.

At the graduate level, I would like to teach advanced courses on type systems and language design. I believe the most effective way to teach such courses is to ask each student to identify a programming problem close to their own research interests—whether in PL or another area—and create a language-based solution. I would also be interested in organizing graduate seminars on modern topics in programming languages, or even to study a textbook or a codebase in depth.

[1] Each One Teach One (E1T1). URL `https://www.eachoneteachone.is`. Accessed 2021-10-12.

[2] Ithaca Media Arts. URL `http://www.ithacamedia.org/`. Accessed 2021-10-12.

[3] Math Explorers' Club. URL `http://pi.math.cornell.edu/~mec/`. Accessed 2021-10-12.

[4] Bootstrap. Data Science. URL `https://www.bootstrapworld.org/materials/data-science/`. Accessed 2021-05-26.

[5] Shandana Mufti. Community service award highlights student's commitment to tutoring, mentorship. URL `https://www.khoury.northeastern.edu/community-service-award-highlights-students-commitment-to-tutoring-mentorship/`. Accessed 2021-10-12.

Diversity Statement

Ben Greenman
2021-11-29

I am committed to diversity and inclusion. It is simply the right thing to do. Any person who is honest and hardworking should have the opportunity to develop their talents as a student or as a researcher.

My own path to a Ph.D was enabled by people who trusted me to succeed in new environments. I moved from a community college to an Ivy league university, and then from a bachelor's degree in labor relations to a summer research position in computer science. Along the way, I was not always confident in my own abilities as a student, TA, or research assistant, but my mentors' support always encouraged me to work a little harder and a little better. One case in particular occurred at a seminar when I was a second-year Ph.D student: During the Q/A period after a talk, my advisor stood up and announced my name and early-stage work to the professionals in the room. This small gesture meant quite a lot to me. If my advisor could believe that I was capable of top-quality research, then I could believe it too.

At Northeastern, I had two opportunities to mentor underrepresented minority students. The first arose when I volunteered with E1T1 [1], an after-school program for inner-city high schools. Twice a week for four months, I met with a Black student who had been failing her AP Java course. It turned out that she was a talented programmer who had simply gotten bored with the curriculum. After some one-on-one review of programming concepts (as opposed to Java particulars), she caught up and ultimately passed her class. The second opportunity came through my Ph.D advisor. An undergraduate who had been studying programming languages with my advisor on weekends wanted to try research. I worked with her to analyze a gradual type system for Python and we eventually co-authored a research paper [2]. This student is now pursuing a Ph.D in programming languages at UCLA.

One of my main goals as a faculty member is to find opportunities to trust others to succeed. When recruiting Ph.D students, I will prioritize sincere and motivated candidates from a variety of backgrounds instead of focusing on those with the highest grades. Once admitted, I will find ways to express my confidence in these students as they work toward a dissertation—in one-on-one conversations, at conferences, and in online forums. In the classroom, I will set clear goals at the beginning of the semester to reduce student anxiety and will foster an environment that is respectful of questions and mistakes. I will promptly grade assignments and respond to private questions. Finally, I will reach out to students in danger of failing to see whether they are in need of special accommodations. It is my hope that these measures help students find the confidence and motivation to succeed, but I will look for more ways to provide support.

Additionally, I believe that my work as a PL researcher can meaningfully address issues with diversity and inclusion. Two major problems that we face today are unintentionally biased algorithms and unrepresentative datasets. These problems are typically viewed from the lens of AI or theory research, but could benefit from language-based solutions. After all, a programming language is the primary interface between human decision-makers and the machines that let people manage tasks at scale. Factors that lead to algorithmic bias could be made manifest in a programming language, giving developers clear feedback about how their algorithm computes. Potential issues in a dataset could be mitigated if domain experts had liguistic support for summarizing, auditing, and labeling data. I will offer topics such as these to all students, underrepresented or otherwise, and follow where their skills and interests lead.

[1] Each One Teach One (E1T1). URL `https://www.eachoneteachone.is`. Accessed 2021-10-12.

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