How Profilers Can Help Navigate Type Migration

BEN GREENMAN*, PLT @ University of Utah, USA
MATTHIAS FELLEISEN, PLT @ Northeastern University, USA
CHRISTOS DIMOULAS, PLT @ Northwestern University, USA

Sound migratory typing envisions a safe and smooth refactoring of untyped code bases to typed ones. However, the cost of enforcing safety with run-time checks is often prohibitively high, thus performance regressions are a likely occurrence. Additional types can often recover performance, but choosing the right components to type is difficult because of the exponential size of the migratory typing lattice. In principal though, migration could be guided by off-the-shelf profiling tools. To examine this hypothesis, this paper follows the rational programmer method and reports on the results of an experiment on tens of thousands of performance-debugging scenarios via seventeen strategies for turning profiler output into an actionable next step. The most effective strategy is the use of deep types to eliminate the most costly boundaries between typed and untyped components; this strategy succeeds in more than 50% of scenarios if two performance degradations are tolerable along the way.

CCS Concepts: • **Software and its engineering** → **Semantics**; Constraints; Functional languages.

Additional Key Words and Phrases: gradual typing, migratory typing, rational programmer, profiling

ACM Reference Format:
Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. How Profilers Can Help Navigate Type Migration. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 241 (October 2023), 30 pages. https://doi.org/10.1145/3622817

1 TYPE MIGRATION AS A NAVIGATION PROBLEM

Sound migratory typing promises a safe and smooth refactoring path from an untyped code base to a typed one [Tobin-Hochstadt and Felleisen 2006; Tobin-Hochstadt et al. 2017]. It realizes the safe part with the compilation of types to run-time checks that guarantee type-level integrity of each mixed-typed program configuration. Unfortunately, these run-time checks impose a large performance overhead [Greenman et al. 2019b], making the path anything but smooth. This problem is particularly stringent for deep run-time checks [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006], but it also applies to shallow run-time checking [Greenman and Migeed 2018]. While improvements to deep and shallow can reduce the severity of the problem, in particular JIT technology for shallow [Roberts et al. 2019; Vitousek et al. 2019], the core issue remains—some configurations need more expensive checks than others.

Greenman [2020, 2022] presents evidence that deep and shallow checks actually come with complementary strengths and weaknesses. Deep checks impose a steep cost at boundaries between typed and untyped code, yet as the addition of types eliminates such boundaries, they enable

*Research done at Brown University

Authors' addresses: Ben Greenman, PLT @ University of Utah, Salt Lake City, Utah, USA, benjaminlgreenman@gmail.com; Matthias Felleisen, PLT @ Northeastern University, Boston, Massachusetts, USA, matthias@ccs.neu.edu; Christos Dimoulas, PLT @ Northwestern University, Evanston, Illinois, USA, chrdimo@northwestern.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2023 Copyright held by the owner/author(s).
2475-1421/2023/10-ART241
https://doi.org/10.1145/3622817

type-driven optimizations that can offset some of the cost [St-Amour 2015]—and sometimes all of it. By contrast, shallow checks impose a low cost at boundaries, but the addition of types almost always increases the overall number of checks. Hence, Greenman argues that developers should, in principle, be able to mix and match deep and shallow checking to get the best-possible type checking benefits with a tolerable performance penalty. Initial empirical data is promising: with the right mixture of checks, it is possible to avoid order-of-magnitude slowdowns that come from either deep or shallow checks alone. Finding a "right" mixture, however, presents a challenge because there are exponentially many possibilities to choose from. Whereas in a purely deep (or shallow) checking scheme, developers have 2^N configurations to choose from, with deep and shallow combined there are 3^N possibilities because each of the N components in the program can be untyped, deep-typed, or shallow-typed.

The large search space raises the following question:

> *How to navigate the 3^N migration lattice of a code base from a configuration with unacceptable performance to one with acceptable performance?*

Since this is a performance problem, a plausible answer is to use profiling tools. But, this conventional response merely refines the above question in two ways, namely:

- *How to use feedback from various profiling tools to choose a next step; and*
- *Whether a sequence of choices leads to a configuration with acceptable performance.*

Such questions call for an empirical investigation. A user study is a viable way forward, but recruiting a large number of people to debug problems in unfamiliar code is costly and introduces confounding factors. Until recently, however, there was no other way to proceed systematically. Instead, this paper reports on the results of a *rational programmer* experiment [Lazarek et al. 2021, 2023, 2020]. The rational programmer method employs algorithmic abstractions (*strategies*) that are inspired by methods that actual humans can follow and that reify a falsifiable hypothesis about one way of using profiling tools and interpreting their feedback. Because the strategies are algorithms, it is straightforward to apply them to thousands of debugging scenarios and test whether they improve performance. In sum, the rational programmer experiment enables a systematic comparison of different ways that human developers[1] might interpret profiler feedback. The winning strategies merit further study, while the losing ones can be set aside.

In short, this paper makes three contributions:

- At the technical level, the rational programmer experiment presents the most comprehensive and systematic examination of type migration paths to date. As such it goes far beyond Greenman [2022]'s prior work. The experiment evaluates 17 different strategies for interpreting profiling output on more than one hundred thousand scenarios using the GTP benchmarks [Greenman 2023]. It yields 5GB of performance and profiling data, which is available online [Greenman et al. 2023a].
- At the object level, the results of the rational programmer experiment provide guidance to developers about how to best use feedback from profilers during type migration. The winning strategy identifies the most expensive boundary and migrates its components to use deep types. This result is a *surprise* given Greenman [2022]'s preliminary data, which implies that combinations of shallow and deep types should lead to the lowests costs overall.
  - Hence, the results also inform language designers about performance dividends from investing in combinations of deep and shallow types.

[1] To distinguish between humans and the rational programmer, the paper exclusively uses "developer" for human coders.

- At the meta level, this application of the rational programmer method to the performance problems of type migration provides evidence for its versatility as an instrument for studying language pragmatics.

The remainder of the paper is organized as follows. Section 2 uses an example to explain the problem in concrete terms. Section 3 introduces the rational programmer method and shows how its use can systematically evaluate the effectiveness of a performance-debugging strategy. Section 4 translates these ideas to a large-scale quantitative experiment. Section 5 presents the data from the experiment, which explores scenarios at a module-level granularity in Typed Racket. Section 6 extracts lessons for developers and researchers. Section 7 places this work in the context of prior research. Section 8 puts this work in perspective with respect to future research.

2 NAVIGATING THE DEEPS AND SHALLOWS BY PROFILING

Over the years, developers have created many large systems in untyped languages. In the meantime, language implementors have created gradually typed siblings of these languages. Since developers tend to enjoy the benefits of type-based IDE support and a blessing from the type checker, they are likely to add new components in the typed sibling language. Alternatively, when a developer must debug an untyped component, it takes a large mental effort to reconstruct the informal types of fields, functions, and methods, and to make this effort pay off, it is best to turn the informal types into formal annotations. In either case, the result is a mixed-typed software system with components that have types and parts that do not.

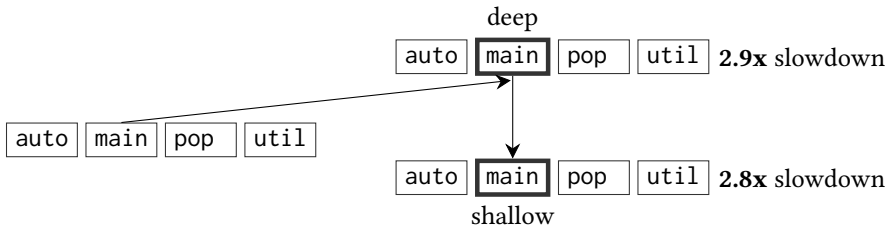
In a sound gradual language, the enforcement of types inflicts a performance penalty. Among the several enforcement approaches that do not limit expressiveness [Greenman et al. 2023b],[2] the two leading ones are deep and shallow types:

- Deep types use higher-order contracts to monitor the boundaries between typed and untyped components [Findler and Felleisen 2002; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Higher-order contracts impose many kinds of performance penalties: they traverse compound values; they wrap higher-order values with proxies to delay checks; and they raise memory consumption due to the proxies' allocation. If there are few boundaries, however, then deep types impose few costs and type-driven optimizations may exceed the performance of the untyped code base [Greenman et al. 2019b].
- Shallow types do not explicitly enforce types at boundaries but delegate checking to tag-level assertions injected at compile-time at strategic places in typed components. Shallow's assertions ask simple questions (is this a list?) and never allocate proxies [Vitousek et al. 2014, 2017]. Each check is inexpensive, but the lack of proxies blurs the boundary between typed and untyped components and leads to a conservatively high number of checks. Suppose a typed function expects a callback. To account for the case that the callback is supplied by an untyped component, every call needs a result check around it to ensure soundness—even if most calls are safe. In general, the addition of more shallow types can lead to more checks.

In either case, the performance penalty can become too high. If so, the developer faces a performance-debugging scenario.

To make these ideas concrete, consider the fsm program from the GTP benchmark suite [Greenman 2023; Greenman et al. 2019b]. The program is the creation of Nguyen and Andreozzi [2016], economists interested in simulating an economy of agents with deterministic strategies. Figure 1a shows the outline of the four-module program: `auto` implements state machines; `pop` coordinates among machines; `main` drives the simulation; and `util` provides helper functions. Focusing on

[2] Nom [Muehlboeck and Tate 2017] and Static Python [Lu et al. 2023] have low-cost but restrictive checks.



(a) Adding deep or shallow types to one fsm module degrades performance

```
  Total cpu time observed: 1192ms (out of 1236ms)
  Number of samples taken: 23 (once every 52ms)

================================================
                              Caller
 Idx   Total       Self      Name+src
       ms(pct)     ms(pct)    Callee
================================================
                              ??? [12]
                              evolve [17]
[17]  818(68.6%)    0(0.0%)  evolve main
                              evolve [17]
                              shuffle-vector [19]
                              death-birth [18]
                              ??? [20]
------------------------------------------------
                              match-up* [22]
                              shuffle-vector [19]
[24]  152(12.7%)  152(12.7%) contract-wrapper
------------------------------------------------
```

(b) Statistical profiler output for the top-right variant

```
cpu time: 984 real time: 984 gc time: 155
Running time is 18.17% contracts
253/1390 ms

(interface:death-birth pop main)
  142 ms
  (->* ((cons/c (vectorof automaton?)
                (vectorof automaton?))
        any/c)
       (#:random any/c)
       (cons/c (vectorof automaton?)
               (vectorof automaton?)))
(interface:match-up* pop main)
  81.5 ms
  (-> ....)
(interface:population-payoffs pop main)
  29 ms
  (-> ....)
```

(c) Boundary profiler output for the same variant

Fig. 1. Profiling during type migration

just the modules of this program suffices because the migration granularity in Typed Racket is by module (each module can be typed or untyped).

The variant of fsm on the left of figure 1a is untyped. If a developer adds deep types to the `main` module, performance is significantly degraded. The mixed-typed variant runs almost three times ($3x$) slower than the untyped one. Switching to shallow types is a one-line change to the module language, but does not remedy the situation. At this point, the question is how to recover the performance of the untyped variant. Each results in different kind of costs

- One option is to roll back the addition of types.
- For developers who prefer typed code and dislike undoing the effort of adding types, a second option is to add (deep or shallow) types to a random module connected to `main`—following a "hunch" like developers sometimes do—but doing so can easily make things worse. For example, if the choice were the `auto` module with shallow types, then performance would degrade further (a 9x slowdown, to be precise).
- If the developer adds deep types to every module, then fsm has no type boundaries and gets the full benefit of optimizations. Performance improves over the untyped variant. However, such a choice represents a heavy migration effort, which a developer who simply wishes to fix `main` and deploy again may be reluctant to invest.

None of these options are compelling. Informed feedback is clearly needed for a solution that recovers performance with a reasonable effort and without discarding types.

The natural choice is to reach for a profiling tool to determine the source of the slowdown. Racket fortunately comes with two such tools:

- a traditional *statistical profiler*, which identifies the time spent in applications; and
- a *boundary profiler*, which attributes the cost of types-as-contracts to specific module boundaries [Andersen et al. 2019; St-Amour et al. 2015].

Both tools are potentially useful and potentially limited due to the mechanics of deep and shallow types. Specifically, the contract-based enforcement of deep types should be a good match for the boundary profiler but not for the statistical profiler. In contrast, shallow checks should favor the statistical but not the boundary profiler. For example, the function below averages a list of numbers. While the total run-time costs of deep or shallow types are comparable for this function, those costs arise in different ways:

```
(: avg (-> [Listof Real] Real))         ;; deep: enforce type as a contract
(define (avg l) (/ (sum l) (length l))) ;; shallow: rewrite code with checks
```

- With deep types, the function gets wrapped in a proxy at the boundary between `avg` and its untyped clients. The proxy checks that clients send only lists that contain only real numbers. The *boundary profiler* is well-suited to discover if these checks are expensive because it attributes costs directly to proxies. Conversely, the statistical profiler is less likely to be useful because it breaks down cost by application. It may, however, discover the costs indirectly if the proxy slows down calls to functions that, in turn, call `avg`.
- With shallow types, the compiler rewrites the body of `avg` to check that its clients send only lists. This check does not examine list elements, but the helper function `sum` will check elements as it accesses them. Because there are no contracts and explicit boundaries in the shallow version, only simple inlined checks, the boundary profiler cannot measure the cost of the types. The *statistical profiler* is in a much better position to find costs because they arise from extra code in the function.

Back to fsm, the bottom half of figure 1 shows the output of the statistical profiler and the boundary profiler for the top-right variant in figure 1a where `main` has deep types.

Statistical profiler. Figure 1b lists two rows from the statistical profiler; the full output has 28 rows. The first row, labeled `[17]`, covers a large percentage (`68.6%`) of the total running time, and it refers to a function named `evolve`, which is defined in the `main` module. The line suggests that calls from `evolve` to other functions account for a high percentage of the total cost. The second row, labeled `[24]`, says that a contract wrapper accounts for a significant chunk (`12.7%`) of the running time. The caller of this contract, from row `[19]` (not shown) is the function `shuffle-vector` from the `pop` module. Putting these clues together, the profiling output indirectly points to the boundary between `main` and `pop` as a significant contributor to the overall cost.

This conclusion, however, is one of many that could be drawn from the full statistical profiler output. Functions from the `util` module also appears in the output, and may be more of a performance problem that those from the `pop` module. Equally unclear is whether the column labeled `Total` is a better guide than the column labeled `Self` or vice versa. High total times point to a context that dominates the expensive parts. High self times point to expensive parts, but these costs might be from the actual computation rather than the overhead of type-checking.

Boundary profiler. Figure 1c shows nearly-complete output from the boundary profiler; only two contracts are omitted. This profiling output attributes `18.17%` of the total running time to contracts, specifically, to the contracts on the three functions whose names begin with an `interface:` prefix.

This output indicates that proxies are wrap untyped functions that flow into typed components. The modules involved are `main` and `pop`. Since `pop` is the untyped one, the hint is to type it.

Adding types to `pop` does improve performance. Concretely, this variant suffers from a 1.2x slowdown. If this overhead is acceptable, the developer is done; otherwise, the search must continue with another round of profiling, searching, and typing.

Summary. At first glance, the effort of eliminating a performance problem seems straightforward. Several factors complicate the search. First, a developer has two typing options not just one. Second, the output from profiling tools is complex. Even for this small program, the statistical profiler outputs 100 lines; identifying the next step is hard. Finally, adding types to the profiler-identified module may degrade the performance even more, in which case the developer may wish to give up. In sum:

> *Navigating a migration lattice with 3^N program configuration is a non-trivial effort, and developers deserve to know how well profiling tools help with this effort.*

3 A RATIONAL APPROACH TO NAVIGATION

When a performance-debugging scenario arises, the key question is *how to modify the program* to improve performance. Profiling tools provide data, but there are many ways to interpret this data. The rational programmer method proceeds by enumerating possible interpretations and testing each one independently.

To begin, the type-migration lattice suggests two general ways to modify a code base: add types to an untyped component, or toggle the types of a typed one from deep to shallow or vice versa. The next question is which component to modify. Since profiling tools identify parts of the code base that contribute to performance degradation, the logical choice is to rank them using a relevant, deterministic order and modify the highest-priority one.

Stepping back, these two insights on modifications and ordering suggest an experiment to determine which combinations of profiling tool, ordering, and modification strategy help developers make progress with performance debugging. To determine the best combination(s), developers must work through a large and diverse set of performance-debugging scenarios. The result should identify successful and unsuccessful strategies for ranking profiler output and modifying code. Of course, it is unrealistic to ask human developers to follow faithfully different strategies through thousands of scenarios. An alternative experimental method is needed.

The rational programmer provides a framework for conducting such large-scale systematic examinations. It is inspired by the well-established idea of rationality in economics [Henrich et al. 2001; Mill 1874]. In more detail, a rational agent is a mathematical model of an economic actor. Essentially, it abstracts an actual economic actor to an entity that, in any given transaction-scenario, acts strategically to maximize some kind of benefit. These agents are (typically) bounded rather than perfectly rational to reflect the limitations of human beings and of available information; they aim to *satisfice* [Simon 1947] their goal since they cannot make maximally optimal choices. Analogously, a rational programmer is a model of a developer who aims to resolve problems with bounded resources. Specifically, it is an algorithm that implements a developer's bounded strategy for satisficing a goal, and thereby enables a large-scale experiment. Developers can use the outcomes of an experiment to decide whether "rational" behavior seems to pay off. In other words, a rational programmer evaluation yields insights into the pragmatic value of work strategies.

So far, the rational programmer has been used to evaluate strategies for debugging logical mistakes.[3] This paper presents the first application to a performance problem.

Experiment Sketch. In the context of profiler-guided type migration, a rational programmer consists of two interacting pieces. The first is strategy-agnostic; it consumes a program, measures its running time, and if the performance is tolerable, stops. Otherwise, the program is a performance-debugging scenario and the second, strategy-specific piece comes into play. This second piece profiles the given program—using the boundary profiler or the statistical profiler—and analyzes the profiling information. Based on this analysis, it modifies the program as described above. This modified version is handed back to the first piece of the rational programmer.

There are many strategies that might prove useful. A successful strategy will tend to eliminate performance overhead, though perhaps after a few incremental steps. An unsuccessful strategy will either degrade performance, or fail to reach an acceptable configuration. Testing several strategies sheds light on their relative usefulness. If one strategy succeeds where another fails, it has higher relative value. Of course, the experiment may also reveal shortcomings of the profiling approach altogether—which would demand additional research from tool creators.

4 EXPERIMENT DESIGN

Turning the sketch from section 3 into a large-scale automated experiment requires formal descriptions for both the profiling strategies of the rational programmer and the notion of debugging scenario. As the preceding section discusses, given a scenario, a strategy identifies the next migration step, which should yield either an acceptable program or another performance-debugging scenario. The preceding section also implies that the migration step is one of three possibilities: (1) to add types and to specify their enforcement regime (deep, shallow); (2) to toggle from one regime to another; or (3) to fail to act. Hence it is possible to specify strategies independently of the scenarios per se. Equipped with formal descriptions, it is possible to turn the generic research question of the introduction into questions with a quantitative nature.

Section 4.1 presents the profiling strategies. Section 4.2 characterizes performance-debugging scenarios, which act as starting navigation points, and how a type-based migration is a path through a lattice of program configurations. It also lays out the criteria for successes and failures for strategies. Finally, section 4.3 formalizes the precise experimental questions and the experimental procedure that answers them.

4.1 The Rational Programmer Strategies

Every program P is a collection of interacting components c. Some components have deep types, some have shallow ones, and some are untyped. Independently of their types, a component c_1, may import another component c_2, which establishes a *boundary* between them, across which they exchange values at run time. Depending on the kind of types at the two sides of the boundary, a value exchange can trigger run-time checks, which may degrade performance.

A profiling strategy should thus aim to eliminate the most costly checks in a program. In formal terms, a profiling strategy is a function that consumes a program P and, after determining its profile, returns a set of pairs (c, t). Here t is either *deep* or *shallow*. Each such pair prescribes a modification of P. For instance, if a strategy returns the singleton set with the pair $(c, deep)$, then the strategy points to a new version of P where component c obtains deep types (if necessary) ; if c is typed,

[3]Prior work distinguishes between *strategies* for interpreting data and *modes* of the rational programmer, which combine a strategy and other parameters into an algorithm. Our experiment has only one parameter, the strategy, and therefore the distinction between strategy and mode is unimportant here.

Profiler	Response	Description
boundary	*optimistic*	Uses the boundary profiler to identify the most expensive boundary in the given program. It recommends that both sides of the target boundary obtain deep types.
	conservative	Like *boundary optimistic* but with shallow types for both sides of the target boundary.
statistical (*self*)	*optimistic*	Uses the statistical profiler to identify the component c_1 that contains the application with the highest self time in the given program, and that has a boundary with at least one component c_2 that has stricter types than c_1. It recommends deep types for c_1 and c_2.
	conservative	Like *statistical(self) optimistic*, with shallow types for c_1, c_2
statistical (*total*)	*conservative*	Like *statistical(self) conservative* with *total* in place of *self*
	optimistic	Like *statistical(self) optimistic* with *total* in place of *self*

Fig. 2. How the basic strategies find and respond to slow boundaries

the strategy just requests toggling from shallow to deep. If a strategy's result is the empty set, it cannot figure out how to proceed.

Basic strategies. Figure 2 describes six basic strategies that rational programmers may use. The strategies differ along two levels: how to use profiler data to identify a set of checks and how to modify the program toward lower costs.

At the first level, the basic strategies choose a profiler and (when necessary) an ordering for its output. The profiler is either *boundary* or *statistical* (section 2). With the boundary profiler, the output is a list of boundaries ordered by cost, so there is no need for the rational programmer to choose an ordering. With the statistical profiler, the output is a list of applications each with two types of costs: the *total* time spent during the call including its dependencies, and the *self* time spent in the call not including dependencies. Because both costs are potentially useful, the rational programmers choose between them. Having ordered the applications, these rational programmers must then identify a boundary. They start with the top-ranked application and seek a boundary between the enclosing component and a component with *stricter* types because the types at those boundaries incur run-time checks. Here, deep is stricter than shallow and shallow is stricter than untyped. If the strategy cannot identify such a boundary, it moves to the next-ranked application (again in terms of either *self* or *total* time). If there are no applications remaining, the strategy fails.

At the second level, basic strategies differ in how they migrate the two sides of their target boundary. Strategies that are *optimistic* turn the types at either side of the boundary to deep. This action eliminates the cost of the boundary and enables type-driven optimizations in both components. But, it may also create boundaries to other components in a kind of ripple effect with potentially disastrous costs. By contrast, *conservative* strategies choose shallow types for both sides of the target boundary. The rationale behind this choice is that, if both sides of a boundary have shallow types, the interactions across the boundary cost less than if only one is deep and, at the same time, unlike with *optimistic* strategies, there is no risk of a ripple effect.

Composite strategies. While the basic strategies ignore the cost of writing type annotations for an untyped component, developers do not. Adding types to an entire module in Typed Racket may require a significant effort. Similarly, the likelihood of ripple-effect costs depends on the number of

Profiler	Response	Description
boundary	*cost-aware optimistic*	Splits the boundaries in the given program to those between typed components and the rest. Delegates to *boundary optimistic* to produce a modification for the given program, but ranks boundaries in the first group higher than those in the second group.
	cost-aware conservative	Like *boundary cost-aware optimistic* but it delegates to *boundary conservative*.
	configuration-aware	If less than 50 % of components in the program have types, it delegates to *boundary conservative*. Otherwise, it delegates to *boundary optimistic*.
statistical (*self*)	*cost-aware optimistic*	Separates the typed components that have boundaries with other typed components from the rest of the components in the given program. Delegates to *statistical(self) optimistic* to produce a modification for the given program, but ranks boundaries between components in the first group higher than the rest to determine the most expensive boundary.
	cost-aware conservative	Like *statistical(self) cost-aware optimistic* but it delegates to *statistical(self) conservative*.
	configuration-aware	Like *boundary configuration-aware* but it delegates to *statistical(self)*.
statistical (*total*)	*cost-aware optimistic*	Like *statistical(self) cost-aware optimistic* but it delegates to *statistical(total) optimistic*.
	cost-aware conservative	Like *statistical(self) cost-aware optimistic* but it delegates to *statistical(total) conservative*.
	configuration-aware	Like *boundary configuration-aware* but it delegates to *statistical(total)*.

Fig. 3. Composite strategies use profiler data and current types to form a response

typed components in the program. With few types, the cost of introducing one component with deep types may well be high; with many types, the chance of a ripple effect is probably low. Hence, the experiment includes composite strategies that take into account the types currently in the codebase before choosing how to respond to profile data.

Figure 3 lists these composite strategies. The *cost-aware* strategies rank the cost of boundaries in terms of the labor needed to equip the two components with types in addition to the costs reported by the profiler. They give priority to those boundaries that involve components that are already typed. For those, migration just means toggling their type enforcement regime, which is essentially no labor. The *configuration-aware* strategies use a heuristic to avoid ripple effects. Instead of committing to a type-enforcement regime up front (optimistically or conservatively), they choose shallow when most components are untyped and deep when most are typed.

Baseline Strategies. An experiment must include a baseline, i.e., the building block for a null hypothesis. Since profilers are the focus of this experiment, baselines must be *profiler-agnostic*. If strategies that ignore profiler data do worse than the basic and composite strategies, then feedback from the profiler evidently plays a meaningful role. Otherwise, comparisons among profiler-aware strategies are meaningless.

The results presented in the next section include two *profiler-agnostic* strategies. The first one, *null*, aims to invalidate the null hypothesis with random choices. Specifically, it picks a random boundary with types of different strictness and flips a coin to choose either an *optimistic* or a *conservative* modification to both sides. The second *profiler-agnostic* strategy, *toggling*, is due to Greenman [2022] and serves as a point of comparison to that prior work. It modifies all typed components to use the same checks, deep or shallow, depending on which regime gives the best performance. It never adds types to an untyped component, which means this strategy has only one chance to improve performance.

4.2 Migration Lattices and their Navigation

Gradual type migration is an open and challenging problem [An et al. 2011; Campora et al. 2017, 2018; Castagna et al. 2020; Chandra et al. 2016; Cristiani and Thiemann 2021; Furr et al. 2009a,b; Garcia and Cimini 2015; Jesse et al. 2021; Kristensen and Møller 2017; Malik et al. 2019; Migeed and Palsberg 2019; Miyazaki et al. 2019; Phipps-Costin et al. 2021; Rastogi et al. 2012; Saftoiu 2010; Siek and Vachharajani 2008; Wei et al. 2020; Yee and Guha 2023]. For any untyped component, a migrating developer has to choose practical type annotations from among an often-infinite number of theoretical ones. But, to make a rational programmer experiment computationally feasible, it is necessary to avoid this dimension.

Fortunately, the construction of the corpus of scenarios from a carefully selected set of suitable seed programs can solve the problem. The established GTP benchmarks [Greenman 2023; Greenman et al. 2019b] are representative of the programming styles in the Racket world, and they come with well-chosen type annotations for all their components. Hence, the migration lattices can be pre-constructed for all benchmark programs. It is thus possible to apply a strategy to any performance-debugging scenario (a program with intolerable performance) in this lattice and use the strategy's recommendations to chart a path through the program's migration lattice.

Intuitively, a strategy S attempts to convert a program P_0 into an improved program P_n in a step-wise manner. Each intermediate point P_i from P_0 to P_n is the result of applying the S to the current program. In essence, S constructs a *migration path*, a sequence of programs $P_0, \ldots, P_n$ from a migration lattice. If S cannot make a recommendation at any point along this path, migration halts. The following definitions formalize these points.

The Migration Lattice. All programs P_i are nodes in the *migration lattice* $\mathcal{L}[\![P_t]\!]$ where P_t, is like P_i but all its components have types (either deep or shallow).[4] In other words, a component in P_i may have no types or toggled types compared to P_t. The bottom element of $\mathcal{L}[\![P_t]\!]$ is P_u, the untyped program. The 3^N nodes of $\mathcal{L}[\![P_t]\!]$ are ordered: $P_i < P_j$ if the untyped components in P_j are a subset of those in P_i. Hence the lattice is organized in *levels* of incomparable configurations. Every configuration in the same level has the same set of untyped components but a distinct combination of deep and shallow types for the typed ones. The notation $P_i \leqslant P_j$ denotes that either $P_i < P_j$ or P_i and P_j are at the same level.

A migration path corresponds to a collection of configurations P_i, $0 \leq i < n$, such that $P_i \leqslant P_{i+1}$. This statement is the formal equivalent to the description from the preceding section that strategies

[4] Although there are several possible choices for P_t, each denotes a unique lattice. By contrast, a lattice based an untyped program ($\mathcal{L}[\![P_u]\!]$) is ambiguous without a pre-determined set of types.

either add types to a single previously untyped component or toggle the type enforcement regime of existing typed components. (No strategy, including the agnostic ones, modifies a boundary where both sides are untyped.) In other words, a migration path is a weakly ascending chain in $\mathcal{L}[\![P_t]\!]$.

Performance-debugging scenarios and success criteria. Completing the formal description of the experiment demands answers to two more questions. The first concerns the selection of the starting points for the strategy-driven migrations, i.e., the *performance-debugging scenarios*. Which configurations P_i quality as slow? Since type checks are the source of performance overhead, the appropriate way to measure costs is by comparing P_i to the untyped configuration P_u:

> *Given a migration lattice, a* performance-debugging scenario *is a configuration P such that $slowdown(P, P_u) > T$.*
> – *$slowdown(P, P_u)$ is the ratio of the performance of P over that of P_u*
> – *T signifies the maximum acceptable performance degradation*

The second question is about differentiating successful from failing migrations. Strictly speaking, performance should always improve, otherwise the developer may not wish to invest any more effort into migration. In the worst case, performance might stay the same for a few migration steps before it becomes acceptable:

> *A migration path $P_0 \ldots P_n$ in a lattice $\mathcal{L}[\![P_t]\!]$ is* strictly successful *iff*
> (1) *P_0 is a performance-debugging scenario,*
> (2) *$slowdown(P_n, P_u) \leq T$, and*
> (3) *for all $0 \leq i < n$, $slowdown(P_{i+1}, P_i) \leq 1$.*

To achieve strict success, a strategy must monotonically improve performance.

An alternative to strict success is to tolerate occasional setbacks. Accepting that a migration path may come with k setbacks where performance gets worse, a k-loose success relaxes the requirement for monotonicity k times:

> *A migration path $P_0 \ldots P_n$ in a lattice $\mathcal{L}[\![P_t]\!]$ is* k-loosely successful *iff*
> (1) *P_0 is a performance-debugging scenario,*
> (2) *$slowdown(P_n, P_u) \leq T$*
> (3) *for all $0 \leq i < n$ with at most k exceptions, $slowdown(P_{i+1}, P_i) \leq 1$*
> *equivalently: $k \geq |\{slowdown(P_{i+1}, P_i) > 1 \mid 0 \leq i < n\}|$*

The construction of a k-loose successful migration path allows a strategy to temporarily degrade performance. The constant k is an upper bound on the number of missteps.

A patient developer may tolerate an unlimited number of setbacks:

> *A migration path $P_0 \ldots P_n$ is N-loosely successful if*
> (1) *P_0 is a performance-debugging scenario,*
> (2) *$slowdown(P_n, P_u) \leq T$*

4.3 The Experimental Questions

Equipped with rigorous definitions, it is possible to formulate the research questions precisely:

> Q_X How successful is a strategy X with the elimination of performance overhead?
> $Q_{X/Y}$ Is strategy X more successful than strategy Y in this context?

Answering Q_X boils down to determining the success and failures of X for all performance-debugging scenarios in all available lattices. If, for a large number of scenarios, X charts migration paths that are strictly successful, the answer is positive. Essentially, the large number of scenarios is evidence that when a rational programmer reacts to profiler feedback following X, it is likely to improve performance. Notably, the above description uses the strict notion of success, which sets a

high bar. Hence, the rational programmer not only manages to tune performance at a tolerable level but each suggestion of its strategy brings the rational programmer closer to its target. Swapping the notion of strict success for k-loose success relaxes this high standard, and offers answers to Q_X when allowing for some bounded flexibility in how well the intermediate suggestions of X help the rational programmer. For completeness, the next section also reports the data collected for the notion of N-loose success.

While an answer to Q_X constitutes an evaluation of a strategy X for interpreting profiler feedback in absolute terms, an answer to $Q_{X/Y}$ is about the relative value of X versus some other strategy Y. This second question asks whether the proportion of scenarios in which X succeeds and Y fails is higher that the proportion of scenarios where Y succeeds and X fails. Of course, the answer may not be clear cut as X and Y may perform equally well in most scenarios, or may have complementary success records. But, relaxing the notion of success by different factors k may help distinguish X and Y based on the quality of the feedback they produce.

Importantly, when Y is the *null* strategy and the answer to $Q_X/null$ is positive, then the experiment invalidates its null-hypothesis. Put differently, the success of X is not due to sheer luck but the rational use of profiler feedback.

Summing up, the rational programmer process for answering Q_X and $Q_{X/Y}$ rests on the following experimental plan:

(1) Create a large and diverse corpus of performance-debugging scenarios.
(2) Calculate the migration paths for each strategy for each scenario.
(3) Compare the successes and failures of the strategies.

Isn't Gradual Typing Dead? Although prior work shows that many configurations of the GTP benchmarks run slowly, it does not answer the Q_X and $Q_{X/Y}$ questions—even in the N-loose case. Greenman et al. [2019b] attempt to investigate an N-loose version of Q_X in a 2^N lattice, but severely limit the length of paths. Greenman [2022] consider longer paths, but only those that start from the untyped configuration and end at a fully-typed configuration. Neither study tests whether configurations that have high slowdown can be systematically transformed to ones with acceptable performance (say: $80x \longrightarrow 70x \longrightarrow 20x \longrightarrow 1x$). That said, without the rational programmer method it is by no means clear how to examine such questions in a principled manner.

5 RESULTS

Running the rational-programmer experiment requires a large pool of computing resources. To begin with, it demands reliable measurements for all complete migration lattices. Then, it needs to use the measurements to compute the outcome of navigating the lattices following each strategy starting from every performance-debugging scenario. This section starts with a description of the measurement process (section 5.1). The remaining two subsections (sections 5.2 and 5.3) explain how the outcome of the experiment answers the two research questions from the preceding section.

5.1 Experiment

The experiment uses the v7.0 release of the GTP Benchmarks with small restructurings to help the boundary profiler attribute costs correctly. The restructuring does not affect the run-time behavior of the programs. See appendix A for details. Also, the experiment omits four of the twenty-one benchmarks: zordoz, because it currently cannot run all deep/shallow/untyped configurations due to a known issue;[5] gregor,quadT, and quadU because each has over 1.5 million configurations,

[5] https://github.com/bennn/gtp-benchmarks/issues/46

Table 1. Datasets, their origin, and server details

Dataset	Server	Racket	Typed Racket
dungeon	`c220g2`	`v8.6.0.2 [cs]`	`29ea3c10`
morsecode	`m510`	same	`700506ca` (cherry pick)
other runtime	`c220g1`	same	default
other profile	`m510`	same	default

Server	Site	CPU Speed	RAM	Disk
`c220g1`	Wisconsin	2.4GHz	128GB	480GB SSD
`c220g2`	Wisconsin	2.6GHz	160GB	480GB SSD
`m510`	Utah	2.0GHz	64GB	256GB SSD

which makes it infeasible to measure their complete migration lattices; and sieve because it has just two modules.

Measurements. The ground-truth measurements consist of running times, boundary profiler output, and statistical profiler output. Collecting this data required three basic steps for each configuration of the 16 benchmarks:

(1) Run the configuration once, ignoring the result, to warm up the JIT. Run eight more times to collect cpu times reported by the Racket `time` function.
(2) Install the boundary profiler and run it once, collecting output.
(3) Install the statistical profiler and run it once, collecting output.

With rare exceptions, our running times are stable. Here *stable* means a 95 % confidence interval based on a two-sided t test [Georges et al. 2007] is within 10 % of the sample mean. A total of 420 configurations (0.4 %) did not converge, but are within 35 % of the sample mean. Most of these came from tetris: 388 configs, or 2 % of the tetris lattice.

The large scale of the experiment complicates the management of this vast measurement collection. The 1,277,694 measurements come from 116,154 configurations. Table 1 (top) shows the division of work across servers from CloudLab [Duplyakin et al. 2019]. Each server ran a sequence of measurement tasks and nothing else; no other users ran jobs during the experiment's reservation time. Table 1 (bottom) lists the specifications of the machines used. In total, the results take up 5GB of disk space. Measurements began in July 2022 and finished in April 2023.

For all but two benchmarks, the measurements used a recent version of Racket (v8.6.0.2, on Chez [Flatt et al. 2019]) and the Typed Racket that ships with it. The exceptions are dungeon and morsecode, which pulled in updates to Typed Racket that significantly affected their performance.[6] Fixing these issues was not necessary for the rational programmer experiment per se, but makes the outcome more relevant to current versions of Racket.

Basic Observations. The measurements confirm that the GTP benchmarks are suitable for the rational programmer experiment (table 2). With $T = 1$ as the goal of migration, all but two benchmarks have plenty of performance-debugging scenarios. Going by configurations rather than benchmarks, over 80 % of all configurations are interesting starting points for the experiment.

[6] https://github.com/racket/typed-racket/pull/1282, https://github.com/racket/typed-racket/pull/1316

Table 2. How many of the 3^N configurations have any overhead to begin with?

Benchmark	3^N	% Scenario	Benchmark	3^N	% Scenario
morsecode	81	82.72 %	lnm	729	40.47 %
forth	81	93.83 %	suffixtree	729	98.49 %
fsm	81	76.54 %	kcfa	2,187	92.87 %
fsmoo	81	83.95 %	snake	6,561	99.97 %
mbta	81	88.89 %	take5	6,561	99.95 %
zombie	81	91.36 %	acquire	19,683	99.23 %
dungeon	243	99.59 %	tetris	19,683	95.47 %
jpeg	243	94.65 %	synth	59,049	99.99 %

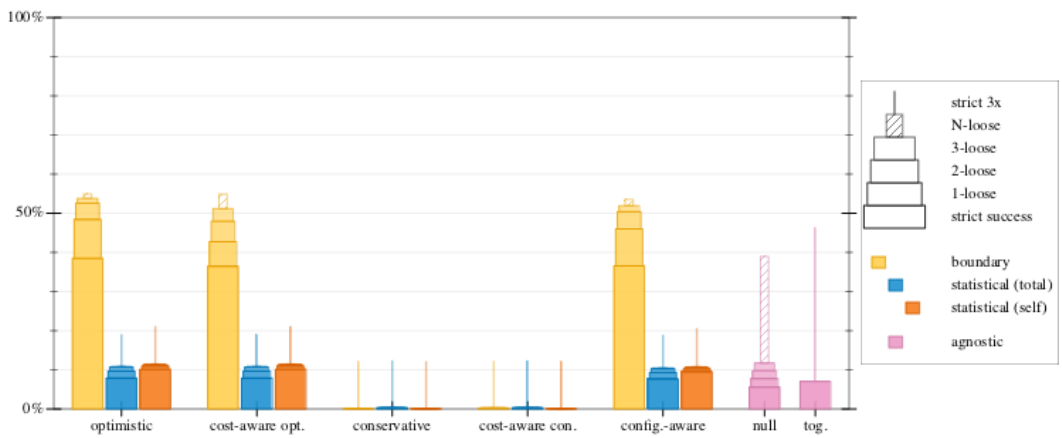


Fig. 4. How many of the 114,428 scenarios does each strategy succeed in, for six notions of success.

5.2 Answering Q_X

Figure 4 presents the results of navigating with all strategies from the preceding section starting from all scenarios. It answers research question Q_X (section 4.3).

Each stacked bar in the "skyline" of figure 4 corresponds to a different strategy. Concretely, it reports the success rate of the strategy for increasingly loose notions of success for $T = 1$. The lowest, widest part of each bar represents the percentage of scenarios where the strategy is strictly successful. The next three levels represent 1-loose, 2-loose, and 3-loose success percentages. The striped spire is for N-loose successes. And finally, the antenna corresponds to a strict success but for $T = 3$. The strategies come with a wide range of success rates:

- *Optimistic* navigation performs well when guided by the *boundary* profiler, finding strict success in almost 40 % of all scenarios. With a 2-loose relaxation, success rises to above 50 %. The results are far worse, however, with *statistical (total)* or *statistical (self)* profiling, both of which rarely succeed.
- *Cost-aware optimistic* is almost as successful as optimistic when driven by *boundary* and equally successful with *statistical (total)* and *statistical (self)*.
- *Conservative* navigation is unsuccessful no matter what profiler it uses.
- *Cost-aware conservative* is unsuccessful as well. Even with N-loose relaxation, it succeeds in very few scenarios (2 %).

- *Configuration-aware optimistic* navigation with *boundary* succeeds in approximately 36 % of all configurations under strict and just over 50 % with 3-loose. With *statistical (total)* and *statistical (self)* profiling, the success rate drops to 10 % even for N-loose.
- *Null* navigation succeeds for roughly 5 % of all scenarios. Though low, this success rate is better than the conservative strategies. Allowing for 1,2,3-loose success improves the rate by small increments. With N-loose, the success rate jumps to nearly 40 %. (These results are the average success rates across three trials. The standard deviations for each number were very low, under 0.10 %.)
- *Toggling* achieves strict success a bit more often than random, for roughly 6 % of all scenarios. The other notions of success do not apply to toggling because it stops after one step.

Antenna: 3x Strict Success. There are two possible reasons for the poor success rate of the conservative strategies. One is that they are entirely unproductive; they lead to worse performance. The other possibility is that they do improve performance but are unable to achieve a Tx overhead because there are no such configurations with mostly shallow types. This second possibility is likely due to the current implementation of shallow types [Greenman 2022], which rarely achieves a speedup relative to untyped code.

To distinguish between these two possibilities, figure 4 includes the antennas that reports strict successes when $T = 3$ is acceptable. The number 3x is the classic, arbitrary Takikawa constant for "acceptable" gradual typing overhead [Bauman et al. 2017; Vitousek et al. 2017]. Changing to 2x or 4x does not significantly change the outcome.

For *conservative* and *cost-aware conservative*, allowing a 3x overhead improves results across the profilers. The strategies succeed in an additional 10 % of scenarios. The optimistic strategies with *statistical* improve in a similar way for 3x success. Optimistic with *boundary* does not improve, and neither does the null strategy. Toggling improves tremendously for 3x success, in line with prior work on shallow, which reports a median worst-case overhead of 4.2x on the GTP Benchmarks [Greenman 2022]. Evidently, about 45 % of configurations can reach a 3x overhead simply by switching to shallow types.

Omitting Hopeless Scenarios. From the perspective of type migration, some scenarios are hopeless. No matter what recommendation a strategy makes for the boundary-by-boundary addition of types to these scenarios, the performance cannot improve to the $T = 1$ goal.

Table 3 lists the number of scenarios in each benchmark and the percentage of hopeful ones. A low percentage in the third column (labeled "% Hopeful") of this table means that the experiment is stacked against any rational programmer. For several benchmarks, this is indeed the case. Worst of all are mbta, dungeon, and take5, which have zero hopeful scenarios. Three others are only marginally better: forth, zombie, and acquire.

Figure 5 therefore revisits the measurements reported in figure 4, focusing on hopeful scenarios only. If there is no migration path from a scenario to a configuration with a tolerable overhead, the scenario is excluded as hopeless. As before, the results for *random boundary* are the average across three runs. The standard deviation is slightly higher than before (< 0.12 %).

For the optimistic strategies, the results are much better. With boundary profiling, they succeed in an additional 10 % of scenarios under either strict or N-loose success. With statistical profiling, the optimistic strategies improve slightly.

Unfortunately, the conservative strategies perform no better when restricted to hopeful scenarios. In fact, the antennae in figure 5 are shorter than the antennae in figure 4. This means that conservative strategies succeeded in the strict 3x sense in a small number of hopeless scenarios that do not appear in figure 5.

Table 3. How many scenarios can possibly reach 1x without removing types?

Benchmark	# Scenario	% Hopeful	Benchmark	# Scenario	% Hopeful
morsecode	67	100.00 %	lnm	295	100.00 %
forth	76	36.84 %	suffixtree	718	100.00 %
fsm	62	100.00 %	kcfa	2,031	100.00 %
fsmoo	68	100.00 %	snake	6,559	100.00 %
mbta	72	0.00 %	take5	6,558	0.00 %
zombie	74	35.14 %	acquire	19,532	5.45 %
dungeon	242	0.00 %	tetris	18,791	100.00 %
jpeg	230	100.00 %	synth	59,046	100.00 %

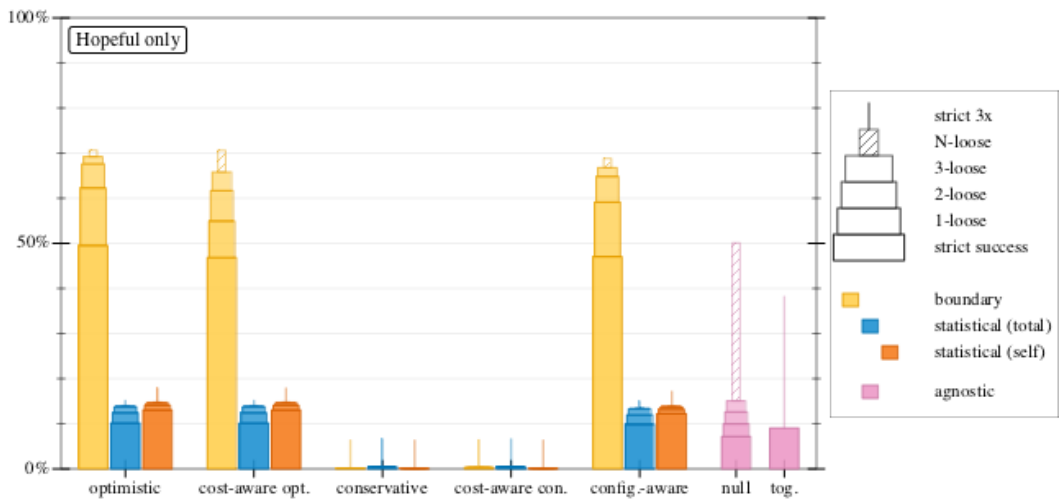


Fig. 5. How many of the 88,992 hopeful scenarios does each strategy succeed in, for six notions of success.

5.3 Answering $Q_{X/Y}$

The preceding subsection hints at how the strategies compare to each other. *Optimistic-boundary* navigation is the most likely to succeed on an arbitrary configuration. *Cost-aware* and *configuration-aware* using the optimistic strategy are close behind. The conservative strategies are least likely to find a successful configuration no matter what profiler they use. Boundary profiling is always more successful than statistical profiling.

However, an unanswered question is whether there are particular cases in which the other strategies succeed and optimistic-boundary fails. Figure 6 thus compares the *optimistic-boundary* strategy to all others, and it thus answers research question $Q_{X/Y}$. The y-axis reports percentages of scenarios. The x-axis lists all strategies including optimistic-boundary (on the left). For each strategy, there are at most two vertical bars. A red bar appears when the other strategy succeeds on configurations where optimistic-boundary fails. A green bar appears for the reverse situation, where optimistic-boundary succeeds but the other fails. Ties do not count, hence the red and green bars do not combine to 100 %.

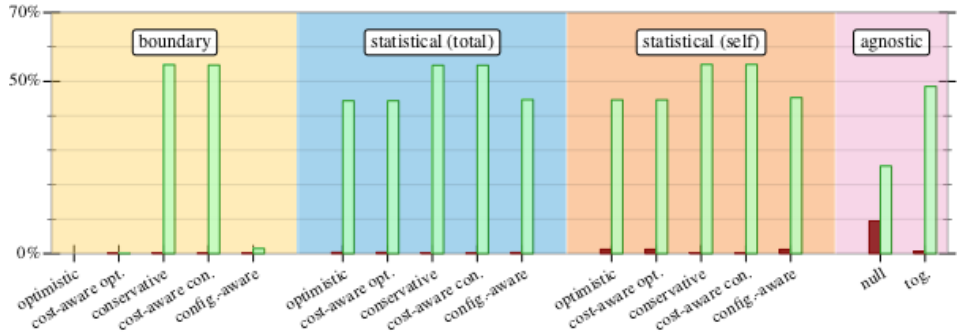


Fig. 6. Boundary optimistic vs. the rest, strict success: losses (red bars) and wins (green bars) on all scenarios.

The tiny red bars and tall green bars give a negative answer to the question of whether optimistic boundary performs worse in certain cases. Other strategies rarely succeed where optimistic-boundary fails.

6 LESSONS FOR DEVELOPERS AND LANGUAGE DESIGNERS

The results of the rational-programmer experiment suggest a few concrete lessons for the developers and also for language designers. Before diving into the details, it is necessary to look at the data for some individual benchmarks (section 6.1). The data is illustrative of general lessons (section 6.2). A closer look at the scenarios yields additional lessons for language designers (section 6.3). Finally, readers should be aware some specific and some general threats to the validity of the data and the conclusions (section 6.4).

6.1 Data from Individual Benchmarks

Figure 4 summarize the successes and failures across all benchmarks. Some of the results for individual benchmarks match this profile well. As figure 7 shows, the tetris and synth are examples of such benchmarks. The two benchmarks share a basic characteristic. They consist of numerous components with a complex dependency graph. Additionally, both benchmarks suffer from a double-digit average performance degradation with deep types [Greenman et al. 2019b].

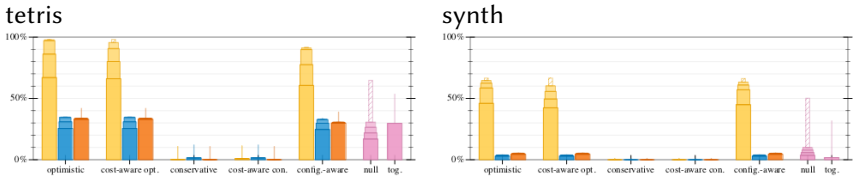


Fig. 7. Examples of migration lattices best navigated with optimistic strategies

For some of the benchmarks, the results look extremely different. The two most egregious examples are shown in figure 8: morsecode and lmn. In contrast to the above examples, these two benchmarks are relatively small and exhibit a rather low worst-case overhead of less than 3x [Greenman 2022].

Finally, some benchmarks exhibit pathological obstacles. Take a look at figure 9, which display the empty plots for mbta and take5. Neither migration lattice of these benchmarks comes with any hopeful performance-debugging scenarios (table 3). Because a developer does not know the

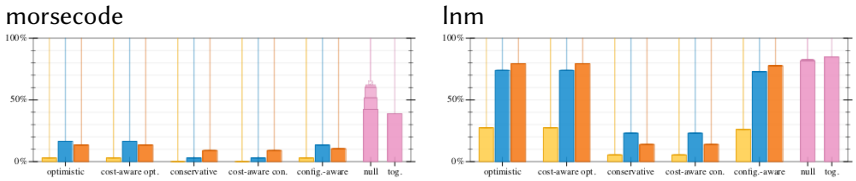


Fig. 8. Examples of migration lattices best navigated with random choices

complete migration lattice and therefore cannot predict whether a scenario is hopeful, general lessons must not depend on the full lattice either.

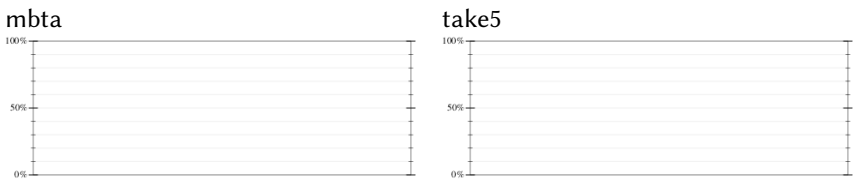


Fig. 9. Empty results for navigations in lattices with zero hopeful scenarios

6.2 General Lessons

Given the general results from the preceding section and the data from the individual benchmarks (see preceding subsection and appendix B), the experiment suggests three lessons for developers and one for language designers.

When a developer faces a performance-debugging scenario, the question is whether to reach for a profiling tool and what kind. The general results and the results for many individual benchmarks give a clear, two-part answer. First, the boundary profiler is superior to the statistical profiler for navigating the migration lattice. Second, this profiler works best on large mixed-typed programs. For small programs with a handful of components and single-digit overheads, the results show that toggling all existing types or randomly choosing a boundary are more effective strategies.

When a developer has reached for the boundary profiler, the next question is how to interpret its feedback. The data implies a single answer. If the boundary profiler is able to identify a particular boundary as a cause of the intolerable performance, the developer is best served by converting both sides of the boundary to use deep types. This modification may prioritize toggling existing shallow types to deep before adding deep types to untyped components. Prioritizing in this order follows from the data for the cost-aware optimistic strategy which is on par with the (cost-unaware) *optimistic* strategy.

When a developer applies an optimistic strategy, configurations along the migration path may suffer from performance problems that are worse than the original ones. In this case, the question is whether the developer should continue with the performance-debugging effort. The data suggests that one setback is a bad sign (10 % of configurations succeed despite one setback) and anything more than two setbacks means that success is highly unlikely. Changing to a different strategy is unlikely to help.

A reader may also wonder whether developers should relax the high standards of eliminating the entire performance overhead. That is, the question is whether a mixed-typed program should run as fast as its (possibly non-existent) untyped variant. But, the antenna data disagrees with

relaxing the standard. With the exceptions of low-overhead programs, if a developer is willing to tolerate a small number of performance degradations along the way, a profiling strategy is as likely to produce a migration path that finds an overhead-free configuration as it is to produce a configuration with some reasonably bounded (3x) overhead.

Language designers can extract a single lesson from the data. The addition of shallow types to the implementation of Typed Racket [Greenman 2022] does not seem to help with the navigation of the migration lattice. All conservative profiling strategies—those that prioritize shallow over deep—yield inferior results compared to optimistic strategies—which prefer deep enforcement. Possibly this is due to the state of profiling technology; no existing profiler may be sufficiently sensitive to detect the aggregate cost of shallow's assertions and point to cost reduction. For now then, language designers are better off investing in deep types and a boundary profiler.

6.3 Specific Lessons

Given that none of the rational programmer strategies succeed on all hopeful scenarios, a first step toward future work is to understand why they fail and whether a modified strategy might succeed. The scenario data provide insights on failures, and the migration lattices show where the opportunities are.

With the boundary profiler, the most common reason that strategies get stuck in that there are no internal boundaries in the output. Either there are no expensive deep type boundaries (the costs may come from shallow types), or the boundaries involve at least one component that lives outside the benchmark in library code. This no-internal issue affects 395,000 scenarios. Roughly one fourth of the scenarios are hopeless at any rate (127K). The rest are hopeful scenarios, and for the vast majority of these (264K) the rational programmer can make one step of progress using statistical profiler data. Adding statistical data as a fallback when no boundary data is available may increase the success rate. This mixed strategy can serve as a starting point for future research.

With statistical profiler data, a huge number of scenarios (745K) get stuck because there are no actionable boundaries in the data. Unfortunately, there are several possible explanations: the boundaries might point to library code, the main costs might point toward essential computations rather than gradual typing checks, or the strategy might fail to upgrade a candidate boundary.

Turning to the migration lattices, table 4 shows where the acceptable configurations are. For a benchmark with N components, it presents a vector with $N + 1$ cells that correspond to the levels in the migration lattice. The leftmost cell represents the untyped configuration, the second-to-left cell represents all $N * 2$ configurations with exactly one typed component, and so on until the rightmost cell, which represents all 2^N fully-typed configurations, Each cell reports the number of acceptable configurations at its level. If this number is zero the cell is red (■), otherwise the cell is green (■). All but a few cells are green, which means that acceptable configurations are spread throughout the lattices. Four benchmarks are exceptional: dungeon and take5 are entirely hopeless, while snake and synth have acceptable configurations only at the endpoints. In the remaining benchmarks, improved strategies have acceptable points to reach for—though, a configuration on level i may need to discard some of its types to reach an acceptable neighbor at the same level.

6.4 Threats to Validity

The validity of the conclusions may suffer from two kinds of threats to their general validity. The first one concerns the experimental setup. The second category is about extrinsic aspects of the rational programmer method.

As for method-internal threats, the first and most important one is that the GTP Benchmarks may not be truly representative of Racket programs in the wild. Several benchmarks are somewhat small with simple dependency graphs and low performance overheads. Since developers typically

Table 4. Which levels of the migration lattice have any acceptable configurations?

Benchmark	#acceptable						Benchmark	#acceptable by lattice level										
morsecode	1	2	4	4	3		lnm	1	9	38	93	138	116	39				
forth	1	2	1	1	0		suffixtree	1	1	0	0	1	4	4				
fsm	1	3	4	7	4		kcfa	1	8	22	33	24	24	29	15			
fsmoo	1	2	4	2	4		snake	1	0	0	0	0	0	0	0	1		
mbta	1	4	4	0	0		take5	1	2	0	0	0	0	0	0	0		
zombie	1	2	3	1	0		acquire	1	8	28	51	45	16	2	0	0	0	
dungeon	1	0	0	0	0	0	tetris	1	12	56	121	169	128	118	133	112	42	
jpeg	1	2	1	1	4	4	synth	1	1	0	0	0	0	0	0	0	0	1

confront performance-debugging scenarios with large, high-overhead programs, they will have to apply the general lessons with some caution. The problem is that such programs may come with large hopeless regions in the migration lattice. Concretely, once a program belongs to the part of the lattice with high performance degradation, no profiling strategy will help the developer escape it. As figure 10 illustrates for the acquire benchmark, the random strategy works better for such large programs than any profiling strategy. It remains an open question how often hopeless regions occur in the wild.

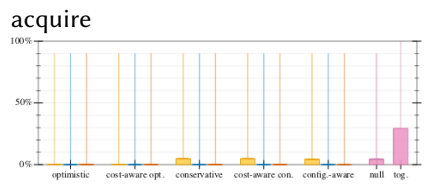


Fig. 10. An example of a large program with a large hopeless region

The second most important internal threat concerns the design of the strategies. While the set of strategies covers the basic approaches to navigation, it is far from complete. For example, certain combinations of the chosen strategies—say, the optimistically cost-aware one with the random one—might deliver better results than pursuing a pure strategy. Another weakness of the strategies is that their migration steps are small. One alternative is to migrate a few modules at a time, similar to the toggling strategy. A second alternative is to split modules into several typed and untyped submodules [Flatt 2013]. On a more technical level, the rational programmers organize statistical profile output by application (see `Idx` in figure 1b) rather than by module. Grouping by module may lead to better recommendations.

A third threat is that the rational programmers reject some configurations that a human developer might accept. If the average overhead of a configuration is within one standard deviation of 1x overhead, the rational programmer accepts it. A handful of configurations lie just outside this cutoff yet within the realm of machine noise [Mytkowicz et al. 2009]; for example, 3.5 % of all configurations are rejected but have an absolute slowdown of at most 100 milliseconds. Accepting these borderline configurations could reduce the number of hopeless scenarios in, say, acquire. However, the 3x "antennas" (see appendix B) include these borderline configurations and nevertheless support our overall conclusions.

Fourth, the large scale of the experiment imposes feasibility constraints on the collected data. Specifically, the experiment collects (and averages) only eight performance measurements per scenario and only one for each profiler.

The design of the experiment attempts to mitigate the method-internal threats. For example, we collected data on single-user machines and confirmed that 99 % of the running times are stable (section 5.1). Still, the reader must keep these threats in mind when drawing conclusions.

As for the method-external threats, the most important one is that the experiment relies on a single language and its eco-system. While this choice is necessary for an apples-to-apples comparison of strategies, it is unclear how the results apply to other language and tool settings. Another aspect of this threat is that the experiment involves only two profilers. While the statistical one is like those found in most language eco-systems, the boundary profiler is unique to Racket. It is possible that other language eco-systems come with profiling tools that might just perform better than those two for some performance-debugging scenarios.

Stepping back, a reader may also question the entire rational-programmer idea as an overly simplistic approximation of performance-debugging work in the real world. But, programming language researchers know quite well that simplified models have an illuminating power. Similarly, empirical PL research has also relied on highly simplified mental models of program execution for a long time. As Mytkowicz et al. [2009] report, ignorance of these simplifications can produce wrong data—and did so for decades. Despite this problem, the simplistic model acted as a compass that helped compiler writers improve their product substantially over the same time period.

Like such models, the rational programmer is a simplified one. While the rational programmer experiment assumes that a developer takes all information into account and sticks to a well-defined, possibly costly process, a developer may make guesses, follow hunches, and take shortcuts. Hence, the conclusions from the rational programmer investigation may not match the experience of developers. Further research that goes beyond the scope of this paper is necessary to establish a connection between the behavior of rational programmers and human developers.

That said, the behavioral simplifications of the rational programmer are analogous to the strategic simplifications that theoretical and practical models make, and like those, they are necessary to make the rational programmer experiment feasible. Despite all simplifications, section 5 demonstrates that the rational programmer method produces results that offer a valuable lens for the community to understand some pragmatic aspects of performance debugging of mixed-typed programs, and it does so at scale and in a quantifiable manner.

7 PRIOR RESEARCH

This work touches a range of existing strands of research. At the object-level, the main motivation for this paper is prior research on the performance issues of sound gradual typing. Two significant sources of inspiration are research on gradual type migration and profiling techniques. At the meta-level, this work builds on and extends prior results on the rational programmer method.

Performance of Gradual Types. Greenman et al. [2019b] demonstrate the grim performance problems of deep gradual types. Adding deep types to just a few components can make a program prohibitively slow, and the slowdown may remain until nearly every component has types. This observation sets the stage for the work in this paper. Furthermore, the experimental approach of that work provides the 3^N migration lattices that are key for the rational programmer experiment herein, and one of the strategies (toggling).

Earlier work observed the negative implications of deep types and proposed mitigation techniques. Roughly, the techniques fall in two groups. The first group proposes the design of alternative run-time checking strategies that aim to control the time and space cost of checks while providing some

type guarantees (e.g. [Greenberg 2015; Greenman et al. 2022; Lu et al. 2023; Rastogi et al. 2015; Richards et al. 2017; Roberts et al. 2019; Siek et al. 2015b, 2009; Swamy et al. 2014; Tsuda et al. 2020]). One notable strategy is transient, which was developed for Reticulated Python [Vitousek 2019; Vitousek et al. 2014, 2019, 2017], adapted to Grace and JIT-compiled to greatly reduce costs [Gariano et al. 2019; Roberts et al. 2019] and later characterized as providing *shallow* types that offer type soundness but not complete monitoring [Greenman et al. 2019a]. Greenman et al. [2023b] provide a detailed analysis and comparison of the overall checking strategy landscape.

The second group of mitigations reduce the time and space required by deep types without changing their semantics [Bauman et al. 2015, 2017; Feltey et al. 2018; Herman et al. 2010; Kuhlenschmidt et al. 2019; Moy et al. 2021; Siek et al. 2015a, 2021]. This is a promising line of work. In the context of Pycket, for example, the navigation problem is easier than Typed Racket because many more configurations run efficiently. But, pathologies still remain. Navigation techniques are an important complement to performance improvements.

Several language designs take a hybrid approach to gradual types so that developers can avoid the costs of deep checks. Thorn and StrongScript use a mixture of *optional* and *concrete* types [Richards et al. 2015; Wrigstad et al. 2010]. Optional types never introduce run-time checks (same as TypeScript [Bierman et al. 2014] or Flow [Chaudhuri et al. 2017]). Concrete types perform cheap nominal type checks but limit the values that components can exchange; for example, typed code that expects an array of numbers cannot accept untyped arrays. Dart 2 explores a similar combination of optional and concrete.[7] Nom [Muehlboeck and Tate 2017, 2021] and SafeTS [Rastogi et al. 2015] independently proposed concrete types as a path to efficient gradual types. Static Python combines concrete and shallow types to ease the limitations of concrete [Lu et al. 2023]. Pyret uses deep checks for fixed-size data and shallow checks for recursive data and functions.[8] Typed Racket recently added shallow and optional types as alternatives to its deep semantics [Greenman 2022].

Gradual Type Migration. Research on gradual type migration can be split in three broad directions: static techniques [Campora et al. 2017; Castagna et al. 2020; Chandra et al. 2016; Furr et al. 2009b; Garcia and Cimini 2015; Kristensen and Møller 2017; Migeed and Palsberg 2019; Phipps-Costin et al. 2021; Rastogi et al. 2012; Siek and Vachharajani 2008]; dynamic techniques [An et al. 2011; Cristiani and Thiemann 2021; Furr et al. 2009a; Miyazaki et al. 2019; Saftoiu 2010], and techniques based on machine learning (ML) [Jesse et al. 2021; Malik et al. 2019; Wei et al. 2020; Yee and Guha 2023]. The dynamic and ML-based techniques exhibit the most scalable results so far as they can produce accurate annotations for a range of components in the wild, such as JavaScript libraries. However, as Yee and Guha [2023] note, the problem is far from solved. Moreover, no existing technique takes into account feedback from profilers to guide migration. One opportunity for future work is to combine the profiling strategies in this paper with migration techniques in the context of automatic or human-in-the-loop tools.

Herder [Campora et al. 2018] estimates relative performance of configurations by combining a static migration technique (variational typing) with a cost semantics. By contrast to our resource-intensive profiling method, Herder is able to find the fastest configuration in several benchmarks without running any benchmark code. However, Herder does not yet handle a full-featured type system (e.g., with union and universal types), and futher experiments are needed to test whether its approximations can find satisficing configurations as well as the best-case one.

Performance Tuning with Profilers. Profilers are the de facto tool that developers use to understand the causes of performance bugs. Tools such as GNU gprof [Graham et al. 1982] established statistical

[7] https://dart.dev/language/type-system#runtime-checks
[8] http://www.pyret.org

(sampling) profilers that collect caller-function execution time data, and paved the way for the development of statistical profilers in many languages, including Racket.

In addition to Racket's statistical profiler, the experiment in this paper also uses Racket's feature-specific profiler [St-Amour et al. 2015]. A feature-specific profiler groups execution time based on (instances) of language features of developers' choosing rather than by function calls. For instance, the boundary profiler that the experiment of this paper employs aggregates the cost of contracts in a program by the boundary that introduces them.

There are two prior works that have used profiler feedback to understand the source of the high cost of gradual types. First, Andersen et al. [2019] show that the Racket feature-specific profiler can detect hot boundaries in programs that use deep types, i.e., it can identify boundaries that are the origin of costly deep checks. Second, Gariano et al. [2019] use end-to-end timing information to identify costly shallow types. We conjecture that using a statistical profile could lead to similar conclusions with fewer runs of the program.

Unlike this paper, prior work on profilers and gradual typing does not examine how to translate profiler feedback to developer actions. The suggested repair is to remove expensive types. In general, most profiling tools do not make recommendations to developers. The Zoom profiler [Patel 2016] was one notable exception, though its recommendations were phrased in terms of assembly language rather than high-level code.

A number of profiling and performance analysis tools provide alternative views. Two recent tools include a vertical profiler [Hauswirth et al. 2004] and a concept-based profiler [Singer and Kirkham 2006]. Both target Java programs. A vertical profiler splits performance data along different levels of abstraction, such as VM cost, syscall cost, and application cost. A concept-based profiler groups performance costs based on user-defined portions of a codebase called concepts [Biggerstaff et al. 1994]. It would be interesting to study alternative profiling and performance analysis techniques in future rational programmer experiments.

The Rational Programmer. Lazarek et al. [2021, 2020] propose the rational programmer as an empirical method for evaluating the role of blame in debugging coding mistakes with software contracts and gradual types. However, the ideas behind the rational programmer go beyond debugging such mistakes. In essence, the rational programmer is a general methodological framework for the systematic investigation of the pragmatics of programming languages and tools. That is, it can quantify the value of the various aspects of a language or a tool in the context of a specific task. In that sense, prior work focuses on a single context: debugging coding mistakes.

This paper shows how the rational programmer applies to experiment design in another context: performance tuning and debugging of performance problem. Hence, it shares the language feature it studies, gradual typing, with prior work. But it looks at a different aspect of its pragmatics. As a result, besides contributing to the understanding of the value of gradual types, it also provides evidence for the generality of the rational programmer method itself.

8 ONWARD!

Sound migratory typing comes with several advantages [Lazarek et al. 2021, 2023] but also poses a serious performance-debugging challenge to developers who wish to use it. Profiling tools are designed to overcome performance problems, but the use of such tools requires an effective strategy for interpreting their output. This paper reports on the results of using the novel rational programmer method to systematically test the pragmatics of five competing strategies that use two off-the-shelf profilers.

At the object level, the results deliver several insights:

(1) The boundary profiler works well if used with any "optimistic" interpretation strategy. That is, developers should eliminate the hottest boundary, as identified by the boundary profiler, by making both modules use deep types.
(2) If a program comes with a low overhead for all mixed-typed variants, the statistical profiler works reasonably well; otherwise the statistical profiler is unhelpful for performance-debugging problems in this context.
(3) While profiling tools help with debugging performance, the data also clarifies that for certain kinds of programs, the migration lattice contains a huge region of "hopeless" scenarios in which no strategy can succeed. These regions call for fundamental improvements to deep and shallow checks.
(4) Finally, the results weaken Greenman [2022]'s report that adding shallow type enforcement is helpful. While toggling to shallow can reduce costs to a 3x overhead in many configurations (figure 4), the poor results for the configuration-aware strategies indicate that it is not a useful stepping stone toward performant (1x) configurations. If parity with untyped code is the goal, deep types are the way to go.

At the meta level, the experiment once again confirms the value of the rational programmer method. Massive simulations of satisficing rational programmers deliver indicative results that clearly contradict anecdotal reports of human developers. As mentioned, a rational programmer is *not* an idealized human developer. It remains an open question whether and how the results apply to actual performance-debugging scenarios when human beings are involved.

Finally, the rational programmer experiment also suggests several ideas for future research. First, the experiment should be reproduced for alternative mixed-typed languages. Nothing else will confirm the value of the optimistic strategy and the boundary profiler. It may also be the case that JIT technology, as demonstrated in Grace [Roberts et al. 2019] and Reticulated [Vitousek et al. 2019], drastically improves the value of the conservative strategy and statistical profiler. Second, the experiment clearly demonstrates that existing profiling tools are not enough to overcome the performance challenges of sound migratory typing. Unless researchers can construct a performant compiler for a production language with sound types, the community must design better profiling tools to guide type migration.

Data Availability Statement. The data for this paper is available on Zenodo, along with scripts for reproducing the experiment and analyzing the results [Greenman et al. 2023a].

ACKNOWLEDGMENTS

Felleisen and Greenman were partly supported by NSF grant SHF 1763922. Greenman also received support from NSF grant 2030859 to the CRA for the CIFellows project. Dimoulas was partly supported by NSF Career Award 2237984. The development of the Racket infrastructure that the paper relies on was supported in part by NSF grant CNS 1823244. Thanks to Cloudlab for hosting the rational programmer experiment. Thanks to Ashton Wiersdorf, Caspar Popova, and Yanyan Ren for feedback on the artifact.

A MODIFICATIONS TO THE GTP BENCHMARKS

To support a rational programmer experiment using boundary profiling, nine of the GTP Benchmarks required a minor reorganization. The change lets the profiler peek through *adaptor modules*, which are a technical device used in the benchmarks. Adaptor modules are a layer of indirection that lets benchmarks with generative types (i.e., Racket structs) support a lattice of mixed-typed configurations [Greenman et al. 2019b; Takikawa et al. 2016]. The following benchmarks required changes: acquire, kcfa, snake, suffixtree, synth, take5, tetris, and zombie.

The trouble with adaptors and profiling is that the name of the adaptor appears in contracts instead of the name of its clients. If one adaptor has three clients, then profiling will attribute costs to one adaptor boundary instead of the three client boundaries. This kind of attribution is bad for the rational programmer because it cannot modify the adaptor to make progress.

The necessary modification is to add client-specific submodules to each adaptor. Taking an adaptor with three clients as an example, the changes are:

(1) define generative types at the top level of the adaptor;
(2) export the generative types *unsafely*, without any contract;
(3) create three submodules, one for each client, each of which imports the generative types, provides them safely, and adapts any other types and functions; and
(4) modify the clients to import from the newly-created submodules rather than the top level.

The submodules do not change run-time behavior, they merely attach client-specific names.

B SKYLINES PER BENCHMARK

Whereas figure 4 reports the overall success rate for every scenario in the experiment, figure 11 separates the results by benchmark. Thus there are 16 plots. Because the benchmarks vary in size from 4 to 10 modules, each plot covers a distinct number of scenarios. Refer to table 3 to see how many scenarios each benchmark has. The colors are from a colorblind-friendly palette [Wong 2011].

Observations.

- The plots for mbta, dungeon, and take5 are empty because none of their scenarios can reach a 1x configuration (table 3).
- The plot for synth is similar to the overall picture (figure 4) because synth has many more scenarios than the other benchmarks. Despite this imbalance, most benchmarks agree with the overall picture. The results for forth, suffixtree, fsm, fsmoo, snake, zombie, tetris, and jpeg all confirm the superiority of the optimistic boundary strategy.
- Both morsecode and lnm do better with statistical profiles than with boundary profiles. These benchmarks have relatively low overhead in their configurations. Boundary profiling therefore reports no information, whereas the statistical profiler can make progress. Curiously, statistical (total) beats statistical (self) in morsecode and the reverse is true in lnm.
- All three profilers do well in kcfa. Boundary profiling is best, but only by a small margin.

C HEAD TO HEAD PER BENCHMARK

Figure 12 compares the *optimistic, boundary* strategy against all the others in each benchmark. Overall these plots support the conclusions from section 5.3.

Observations.

- The plots for mbta, dungeon, and take5 are empty. No strategy ever succeeds.
- The plot for acquire is nearly empty, again because successes are rare. The conservative and configuration-aware strategies with boundary profiles are slightly better than the rest.
- Because boundary profiling tends to get stuck in morsecode and lnm due to low overhead at boundaries, there are noticeable red bars for all strategies that use statistical profiles. Statistical out-performs optimistic boundary profiling. But *null* also does well and even beats statistical in morsecode. This unexpectedly high success of null suggests that profiling is not needed for these benchmarks; better performance is close at hand with any change to the boundaries.

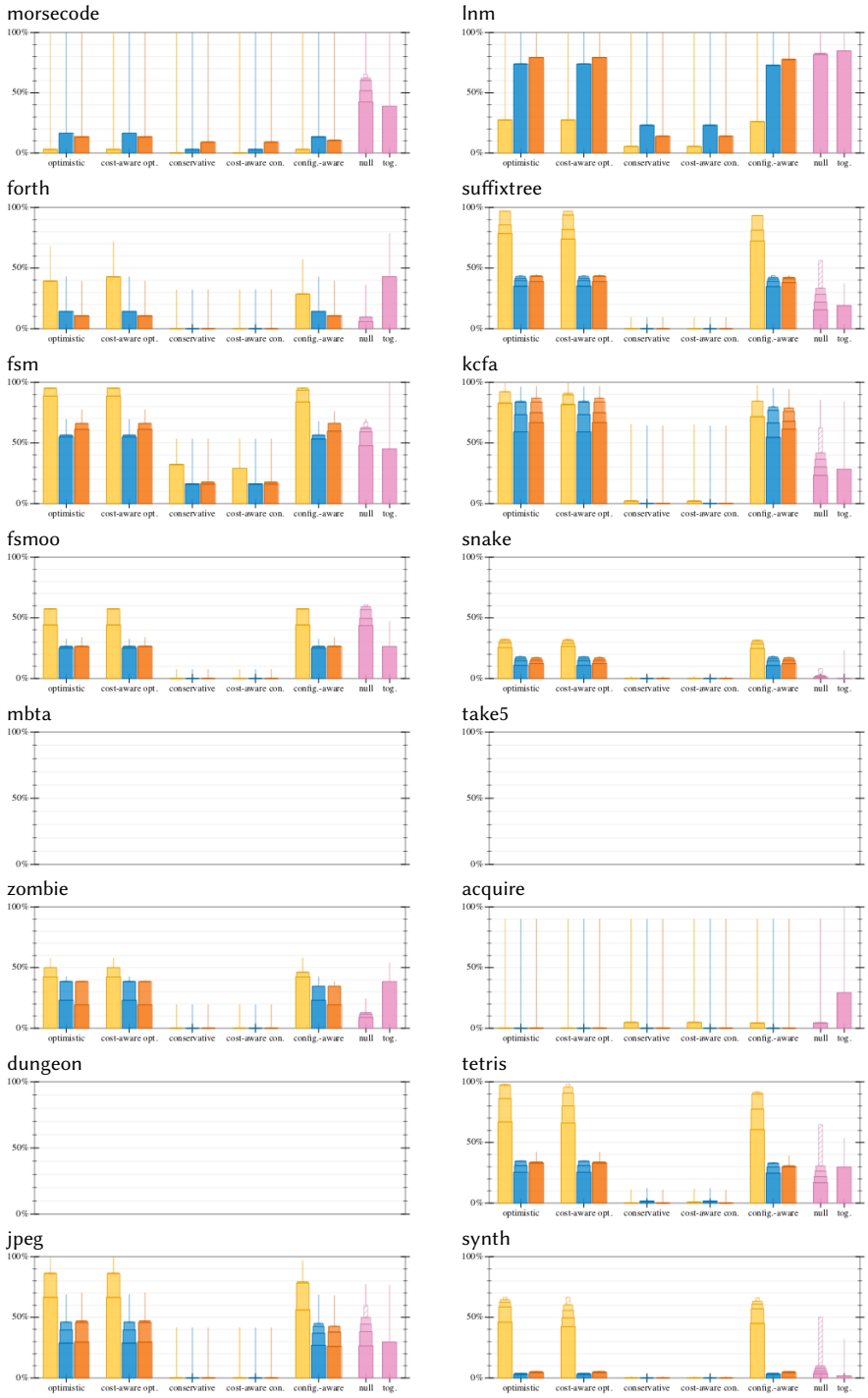


Fig. 11. How scenarios in each benchmark does each strategy succeed in?

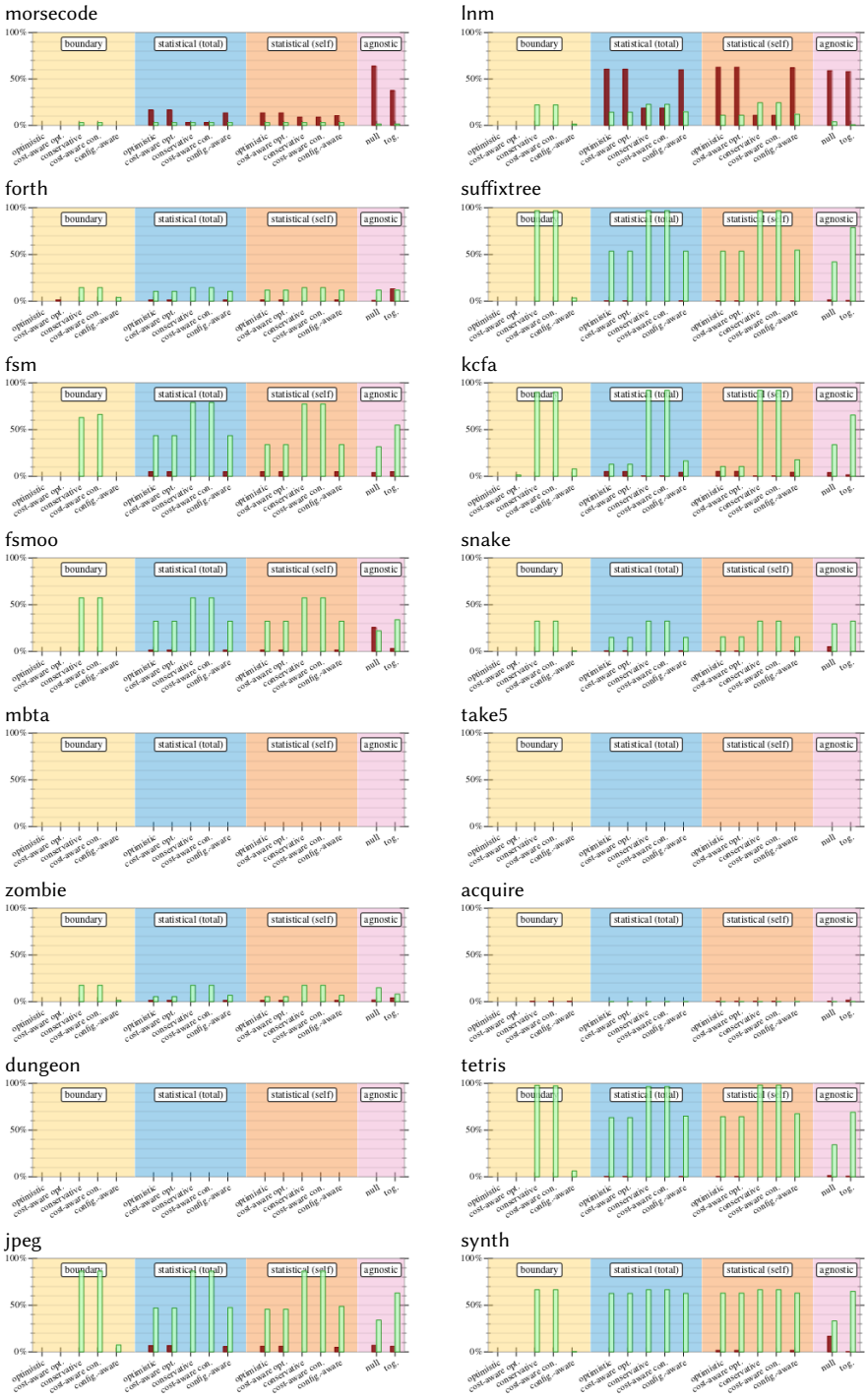


Fig. 12. Optimistic vs. the rest, comparing strict successes in each benchmark.

REFERENCES

Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. 2011. Dynamic Inference of Static Types for Ruby. In *POPL*. 459–472. https://doi.org/10.1145/1926385.1926437

Leif Andersen, Vincent St-Amour, Jan Vitek, and Matthias Felleisen. 2019. Feature-Specific Profiling. *TOPLAS* 41, 1, Article 4 (2019), 34 pages.

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A Tracing JIT for a Functional Language. In *ICFP*. 22–34. https://doi.org/10.1145/2784731.2784740

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: only Mostly Dead. *PACMPL* 1, OOPSLA (2017), 54:1–54:24.

Gavin Bierman, Martin Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP*. 257–281.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program Understanding and the Concept Assignment Problem. *Commun. ACM* 37, 5 (1994), 72–82. https://doi.org/10.1145/175290.175300

John Peter Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2017. Migrating Gradual Types. *PACMPL* 2, POPL, Article 15 (2017), 29 pages. https://doi.org/10.1145/3158103

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *PACMPL* 2, ICFP (2018), 98:1–98:30. https://doi.org/10.1145/3236793

Guiseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2020. Gradual Typing: A New Perspective. *PACMPL* 4, POPL (2020), 16:1–16:32.

Satish Chandra, Colin S. Gordon, Jean-Baptiste Jeannin, Cole Schlesinger, Manu Sridharan, Frank Tip, and Youngil Choi. 2016. Type Inference for Static Compilation of JavaScript. In *OOPSLA*. 410–429. https://doi.org/10.1145/2983990.2984017

Avik Chaudhuri, Panagiotis Vekris, Sam Goldman, Marshall Roch, and Gabriel Levy. 2017. Fast and Precise Type Checking for JavaScript. *PACMPL* 1, OOPSLA (2017), 56:1–56:30.

Fernando Cristiani and Peter Thiemann. 2021. Generation of TypeScript Declaration Files from JavaScript Code. In *MAPLR*. 97–112. https://doi.org/10.1145/3475738.3480941

Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *PACMPL* 2, OOPSLA (2018), 133:1–133:27.

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP*. 48–59.

Matthew Flatt. 2013. Submodules in Racket: you want it when, again?. In *GPCE*. 13–22.

Matthew Flatt, Caner Derici, R. Kent Dybvig, Andrew W. Keep, Gustavo E. Massaccesi, Sarah Spall, Sam Tobin-Hochstadt, and Jon Zeppieri. 2019. Rebuilding Racket on Chez Scheme (experience report). *PACMPL* 3, ICFP (2019), 78:1–78:15. https://doi.org/10.1145/3341642

Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. 2009a. Profile-Guided Static Typing for Dynamic Scripting Languages. In *OOPSLA*. 283–300. https://doi.org/10.1145/1640089.1640110

Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. 2009b. Static Type Inference for Ruby. In *SAC*. 1859–1866. https://doi.org/10.1145/1529282.1529700

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *POPL*. 303–315. https://doi.org/10.1145/2676726.2676992

Isaac Oscar Gariano, Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Which of My Transient Type Checks Are Not (Almost) Free?. In *VMIL*. 58–66.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *OOPSLA*. ACM, 57–76. https://doi.org/10.1145/1297027.1297033

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A Call Graph Execution Profiler. In *CC*. 120–126. https://doi.org/10.1145/800230.806987

Michael Greenberg. 2015. Space-Efficient Manifest Contracts. In *POPL*. 181–194.

Ben Greenman. 2020. *Deep and Shallow Types*. Ph. D. Dissertation. Northeastern University.

Ben Greenman. 2022. Deep and Shallow Types for Gradual Languages. In *PLDI*. 580–593.

Ben Greenman. 2023. GTP Benchmarks for Gradual Typing Performance. In *REP*. ACM, 102–114. https://doi.org/10.1145/3589806.3600034

Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023a. *Artifact: How Profilers Can Help Navigate Type Migration*. https://doi.org/10.5281/zenodo.8136116

Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023b. Typed–Untyped Interactions: A Comparative Analysis. *Transactions on Programming Languages and Systems* 45, 1, Article 4 (2023), 54 pages. https://doi.org/10.1145/3579833

Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2019a. Complete Monitors for Gradual Types. *PACMPL* 3, OOPSLA (2019), 122:1–122:29.

Ben Greenman, Lukas Lazarek, Christos Dimoulas, and Matthias Felleisen. 2022. A Transient Semantics for Typed Racket. *Programming* 6, 2 (2022), 1–25.

Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-Tag Soundness. In *PEPM*. 30–39.

Ben Greenman, Asumu Takikawa, Max S. New, Daniel Feltey, Robert Bruce Findler, Jan Vitek, and Matthias Felleisen. 2019b. How to Evaluate the Performance of Gradual Type Systems. *Journal of Functional Programming* 29, e4 (2019), 1–45.

Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Priented Applications. In *OOPSLA*. 251–269. https://doi.org/10.1145/1028976.1028998

Joseph Henrich, Robert Boyd, Samuel Bowles, Colin Camerer, Ernst Fehr, Herbert Gintis, and Richard McElreath. 2001. In Search of Homo Economicus: Behavioral Experiments in 15 Small-Scale Societies. *American Economic Review* 91, 2 (2001), 73–78.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-Efficient Gradual Typing. *Higher-Order and Symbolic Computation* 23, 2 (2010), 167–189.

Kevin Jesse, Premkumar T. Devanbu, and Toufique Ahmed. 2021. Learning Type Annotation: Is Big Data Enough?. In *ESEC/FSE/*. 1483–1486. https://doi.org/10.1145/3468264.3473135

Erik Krogh Kristensen and Anders Møller. 2017. Inference and Evolution of TypeScript Declaration Files. In *FASE*. 99–115.

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *PLDI*. 517–532.

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2021. How to Evaluate Blame for Gradual Types. *PACMPL* 5, ICFP (2021), 68:1–68:29.

Lukas Lazarek, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. How to Evaluate Blame for Gradual Types, Part 2. *PACMPL* 7, ICFP (2023), 194:1–194:28.

Lukas Lazarek, Alexis King, Samanvitha Sundar, Robert Bruce Findler, and Christos Dimoulas. 2020. Does Blame Shifting Work? *PACMPL* 4, POPL (2020), 65:1–65:29.

Kuang-Chen Lu, Ben Greenman, Carl Meyer, Dino Viehland, Aniket Panse, and Shriram Krishnamurthi. 2023. Gradual Soundness: Lessons from Static Python. *Programming* 7, 1 (2023), 2:1–2:40.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript Function Types from Natural Language Information. In *ICSE*. 304–315. https://doi.org/10.1109/ICSE.2019.00045

Zeina Migeed and Jens Palsberg. 2019. What is Decidable about Gradual Types? *PACMPL* 4, POPL, Article 29 (2019), 29 pages. https://doi.org/10.1145/3371097

John Stuart Mill. 1874. *Essays on Some Unsettled Questions of Political Economy*. Longmans, Green, Reader, and Dyer.

Yusuke Miyazaki, Taro Sekiyama, and Atsushi Igarashi. 2019. Dynamic Type Inference for Gradual Hindley–Milner Typing. *PACMPL* 3, POPL, Article 18 (2019), 29 pages. https://doi.org/10.1145/3290331

Cameron Moy, Phúc C. Nguyundefinedn, Sam Tobin-Hochstadt, and David Van Horn. 2021. Corpse Reviver: Sound and Efficient Gradual Typing via Contract Verification. *PACMPL* 5, POPL, Article 53 (2021), 28 pages. https://doi.org/10.1145/3434334

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *PACMPL* 1, OOPSLA (2017), 56:1–56:30.

Fabian Muehlboeck and Ross Tate. 2021. Transitioning from Structural to Nominal Code with Efficient Gradual Typing. *PACMPL* 5, OOPSLA (2021), 127:1–127:29. https://doi.org/10.1145/3485504

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data without Doing Anything Obviously Wrong!. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 265–276.

Linh Chi Nguyen and Luciano Andreozzi. 2016. Tough Behavior in the Repeated Bargaining Game. A Computer Simulation Study. *EAI Endorsed Trans. Serious Games* 3, 8 (2016), e5. https://doi.org/10.4108/eai.3-12-2015.2262403

Sanjay Patel. 2016. RotateRight Zoom. https://github.com/rotateright/rrprofile

Luna Phipps-Costin, Carolyn Jane Anderson, Michael Greenberg, and Arjun Guha. 2021. Solver-Based Gradual Type Migration. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 111 (2021), 27 pages. https://doi.org/10.1145/3485488

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *POPL*. 481–494. https://doi.org/10.1145/2103656.2103714

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *POPL*. 167–180.

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-Time Knowledge to Optimize Gradual Typing. *PACMPL* 1, OOPSLA (2017), 55:1–55:27.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *ECOOP*. 76–100.

Richard Roberts, Stefan Marr, Michael Homer, and James Noble. 2019. Transient Typechecks are (Almost) Free. In *ECOOP*. 15:1–15:29.

Claudiu Saftoiu. 2010. *JSTrace: Run-time Type Discovery for JavaScript*. Master's thesis. Brown University. https://cs.brown.edu/research/pubs/theses/ugrad/2010/saftoiu.pdf

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *PLDI*. 425–435.

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *ESOP*. 432–456.

Jeremy G. Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *ESOP*. 17–31.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *SFP. University of Chicago, TR-2006-06*. 81–92.

Jeremy G. Siek, Peter Thiemann, and Philip Wadler. 2021. Blame and Coercion: Together Again for the First Time. *Journal of Functional Programming* 31 (2021), e20. https://doi.org/10.1017/S0956796821000101

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-Based Inference. In *DLS*. 7:1–7:12. https://doi.org/10.1145/1408681.1408688

Herbert A. Simon. 1947. *Administrative Behavior*. MacMillan.

Jeremy Singer and Chris Kirkham. 2006. Dynamic Analysis of Program Concepts in Java. In *PPPJ*. 31–39. https://doi.org/10.1145/1168054.1168060

Vincent St-Amour. 2015. *How to Generate Actionable Advice about Performance Problems*. Ph. D. Dissertation. Northeastern University.

Vincent St-Amour, Leif Andersen, and Matthias Felleisen. 2015. Feature-Specific Profiling. In *CC*. 49–68.

Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *POPL*. 425–437.

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *POPL*. 456–468.

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: from Scripts to Programs. In *DLS*. 964–974.

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *SNAPL*. 17:1–17:17.

Yuya Tsuda, Atsushi Igarashi, and Tomoya Tabuchi. 2020. Space-Efficient Gradual Typing in Coercion-Passing Style. 8:1–8:29. https://doi.org/10.4230/LIPICS.ECOOP.2020.8

Michael M. Vitousek. 2019. *Gradual Typing for Python, Unguarded*. Ph. D. Dissertation. Indiana University.

Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for python. In *DLS*. 45–56.

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *DLS*. 28–41.

Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: Open-World Soundness and Collaborative Blame for Gradual Type Systems. In *POPL*. 762–774.

Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. 2020. LambdaNet: Probabilistic Type Inference using Graph Neural Networks. In *ICLR*.

Bang Wong. 2011. Color blindness. *Nature Methods* 8, 6 (2011), 441–442. https://doi.org/10.1038/nmeth.1618

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL*. 377–388.

Ming-Ho Yee and Arjun Guha. 2023. Do Machine Learning Models Produce TypeScript Types That Type Check?. In *ECOOP*. Schloss Dagstuhl, 37:1–37:28. https://doi.org/10.4230/LIPIcs.ECOOP.2023.37