Navigating Mixed-Typed Migration with Profilers

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

The component-by-component migration of a program from untyped to typed can trigger unintended performance degradations. When such a degradation occurs, typing well-chosen components can lessen the cost of type enforcement, while typing poorly-chosen components can exacerbate it. This paper examines whether off-the-shelf profiling tools deliver information that is an effective guide for navigating these migration choices in Typed Racket. Using the rational-programmer method, the paper tests this hypothesis with an experiment that involves tens of thousands of performance-debugging scenarios, two different profiler types, and twenty-three strategies that convert profiler output to migration choices. The most effective strategy relies on a boundary profiler to identify the costliest inter-component boundary, then adds deeply-enforced types to both sides. When the output of this profiler fails to identify a boundary, the strategy takes a random migration step—similar to the way many computational approaches resort temporarily to random choices to avoid getting stuck. The strategy completely eliminates the cost of run-time type checks in more than half of the scenarios if at most two performance degradations are tolerable along the way.

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

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

ACM Reference Format:
Nathaniel Hejduk, Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2024. Navigating Mixed-Typed Migration with Profilers. *ACM Trans. Program. Lang. Syst.* XX, X, Article XX (August 2024), 55 pages. https://doi.org/XXXXXXX.XXXXXXX

1 TYPE MIGRATION AS A NAVIGATION PROBLEM

Sound migratory typing promises a safe and smooth refactoring path from an untyped program to a typed one [75, 76]. It realizes the safe part with the compilation of types to run-time checks that guarantee the type-level integrity of each mixed-typed program configuration. Unfortunately, these run-time checks impose a large performance overhead [34], making the path anything but smooth. This problem is particularly stringent for deep run-time checks [66, 75], but it also applies to shallow run-time checking [33]. While improvements to deep and shallow can reduce the severity of the problem, in particular JIT technology for shallow [61, 80], the core issue remains—some configurations need more expensive checks than others.

Authors' Contact Information: Nathaniel Hejduk, PLT @ Northwestern University, Evanston, Illinois, USA, nathanielhejduk2028@u.northwestern.edu; 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.

Greenman [26, 27] presents evidence that deep and shallow checks 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 type-driven optimizations that can offset some of the cost [71]—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, combining deep and shallow yields 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 program 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. This conventional response generates two new questions:

- *How to use feedback from various profiling tools to choose a next step; and*
- *Whether these profiler-guided steps eventually lead to a configuration with acceptable performance.*

Such questions call for an empirical investigation. A user study is a possible way forward, but recruiting a large number of people to debug problems in unfamiliar code is costly and introduces confounding factors. This paper instead reports on the results of a *rational-programmer* experiment [42–44]. The rational-programmer method employs algorithmic abstractions called *strategies* that draw inspiration from methods that actual humans might follow and that reify falsifiable hypotheses about particular ways 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 [27]. The experiment evaluates 23 different strategies for interpreting profiling output on more than one hundred thousand scenarios using the GTP benchmarks [28].
- 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 picks a boundary and migrates its components to use deep types when possible. In contrast, strategies that migrate towards shallow types are significantly less successful. This result is a *surprise* given Greenman [27]'s preliminary data, which implies that combinations of shallow and deep types should lead to the lowest 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 language designers. Section 7 places this work in the context of prior research. Section 8 puts this work in perspective with respect to future research.

1.1 Relation to prior work

This paper is a radical reworking of a prior publication with a related author set [31]. Importantly, it reports the results of an extended experiment and data analysis that confirm a surprising discrepancy between the measurements of Greenman [27] and the comprehensive measurements in the prior work.

In detail, this paper differs from Greenman et al. [31] along three dimensions:

(1) In terms of strategies:
  - ☞ This paper features a new family of so-called *second-chance* strategies that significantly outperform those of Greenman et al. [31]—see Section 4.
  - ☞ This paper replaces the strategies of Greenman et al. [31] that favor shallow checks with improved variants that avoid unnecessary costs—see Section 4.1.
  - ☞ Greenman et al. [31]'s *null* strategy, which serves as a baseline, depends on two sources of randomness: (1) a mock profiler that returns a randomly chosen boundary and (2) random alternation between a *deep* update strategy and a *shallow* one. The results from Greenman et al. [31] show *shallow* update strategies to be generally unsuccessful, which makes *null*'s dependence on one an unfair disadvantage. In order to untangle the effects of shallow-vs-deep from the effects of profiler-vs-no-profiler, this paper replaces the *null* strategy with two strategies, *random deep* and *random shallow*. Both of these new random strategies keep (1) in place but commit to only deep checks or only shallow checks up front—see Section 4.1.

(2) In terms of data analysis:
  - ☞ This paper fixes a methodological mistake with the data analysis of Greenman et al. [31]. That analysis aggregates the experimental results from all benchmarks, treating each performance-debugging scenario with equal weight. As a result, a couple of large benchmarks that provide many more performance-debugging scenarios than the others dominate the conclusions of the overall experiment. In response, the analysis in this paper normalizes each benchmark's results by the total number of scenarios in the benchmark's lattice before aggregating—see Section 4.3.
  - ☞ Greenman et al. [31] runs *null*, its only nondeterministic strategy, three times per scenario and divides by three to get the average success rate. To reduce the effect of random chance on the results, we up the number of times each nondeterministic strategy is run per scenario to 10—see Section 5.2.
  - ☞ We add a new research question $Q_M([BP.D])$ examining *cost-aware*ness—see Sections 4.3 and 5.4.
  - ☞ While both Greenman et al. [31] and this paper focus on whether strategies achieve performance parity with the untyped versions of the benchmarks (1x overhead), both include alternative data about how the strategies fare if some overhead is tolerable. Greenman et al. [31] provide an analysis of their strategies'

   success when the threshold of acceptable overhead for a configuration is an unrealistic 3x compared to its corresponding untyped configuration. In contrast, this paper reports on more realistic alternative thresholds: 1.1x and 1.5x—see Section 5.6.

   ☞ This paper crowns the strategy with the most reliably positive success rate as the "recommended" strategy and offers a closer look at its behavior throughout the migration process, broken down by benchmark—see Section 5.5.

   ☞ This paper considers a previously overlooked potential threat to the validity of the experimental results. Namely, migration lattices naturally have many more highly-typed configurations than configurations with few typed components. This unbalanced nature of the lattice challenges the relevance of the experiment's results in early stages of migration, when most components are untyped. A new analysis shows that restricting the experiment's starting points to these early stages does not affect the general trends of the results—see Section 5.7.

(3) In terms of results:

   ☞ Due to the decoupling of the *random deep* and *random shallow* strategies and the fairer approach to benchmark weighting, *boundary deep*, which Greenman et al. [31] found to be the best strategy, no longer outperforms every baseline strategy. The *random deep* strategy ends up with a success rate higher than *boundary deep*'s. However, *random deep* is not the best either; a couple of the new *second-chance* strategies end up on top—see Section 5.2.

In sum, this paper both adds to and replaces parts of the material from the prior work. The remainder of the paper points out material that has been altered with dedicated notes marked by the ☞symbol.

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 might 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 often best to turn the informal types into formal annotations. In either case, the result is a mixed-typed software system where some components have types, and others do not.

In a sound mixed-typed language, the enforcement of types inflicts a performance penalty. Among the several enforcement approaches that do not limit expressiveness [29],[2] the two leading ones are deep and shallow:

- The deep approach uses higher-order contracts to monitor the boundaries between typed and untyped components [18, 66, 75]. 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 these proxies' allocation. If there are few boundaries, however, then deep types impose few costs, and type-driven optimizations may push the performance to be better than that of the untyped program [34].
- The shallow approach does not explicitly enforce types at boundaries. Rather, it delegates checking to tag-level assertions injected at compile-time into strategic places in typed components. Shallow's assertions ask simple questions (is this a list?) and never allocate proxies [79, 81]. Each check is inexpensive, but the lack of proxies

[2] Nom [51] and Static Python [45] have low-cost checks that necessitate restricting the expressive power of the language.

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, each addition of shallow types to a program leads to more checks and thus a higher performance cost.

To make these ideas concrete, consider the Racket code below, which defines a function that 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))
(define (avg l) (/ (sum l) (length l)))
(: sum (-> [Listof Real] Real))
(define (sum l) (if (empty? l) 0 (+ (first l) (sum (rest l)))))
```

- When avg and sum have deep types, avg gets wrapped in a proxy at the boundary between avg and its untyped clients. The proxy verifies that clients send only lists that contain only real numbers. No such proxy wraps sum at the boundary between sum and avg, since the static checks of the type system verify that avg always sends values of the correct type to sum.
- When avg and sum have shallow types, the compiler adds a check to the body of avg verifying that l is a list, a check to the body of sum verifying that l is a list, and a check to the body of sum verifying that the result of the first operation is a real number.

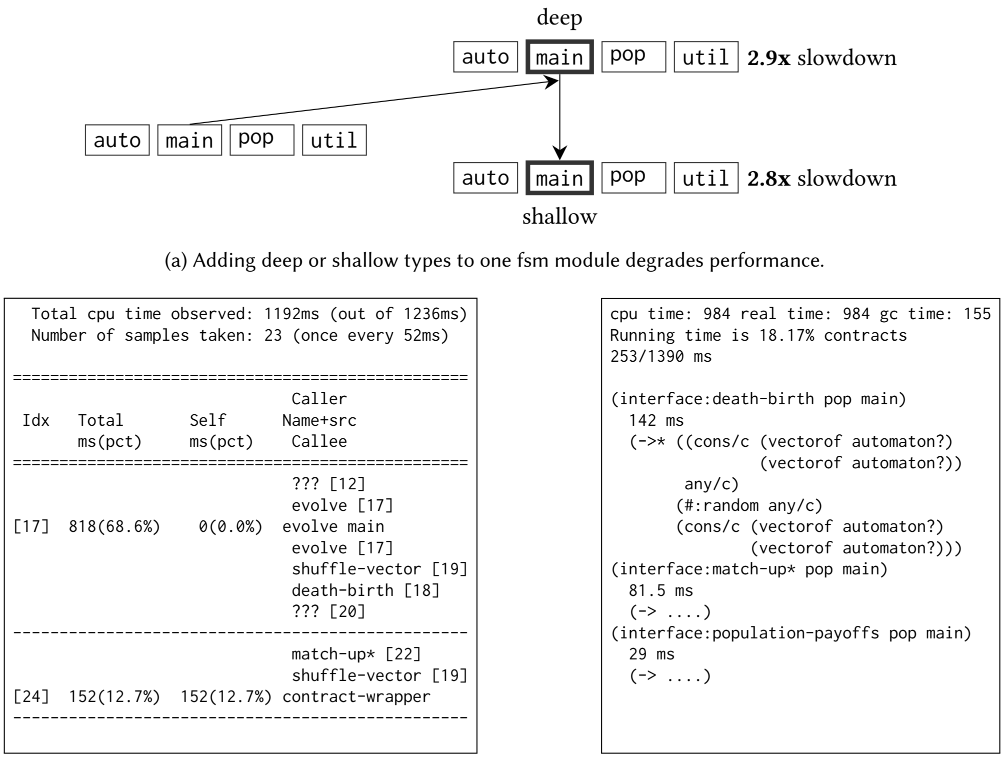
Notice that in the case of deep types, there is a single check verifying that l is a list, but in the shallow case, there are two. This is because shallow-typed components do not offer any guarantees about the values they pass to other components, so each component must check at run time the types of everything it receives from another component. A shallow-typed component's checks are designed to protect only itself, while a deep-typed component offers guarantees to other deep-typed components about every value that enters its domain. When deep types are in use, the contract on avg checks that the type associated with l is valid for the actual value passed into avg before any part of avg starts to be evaluated, hence the guarantee that the value of l passed to sum is of the correct type. Deep checks manifest as a high cost up front (at the type boundary) in order to avoid costs within the typed code, whereas shallow checks have low costs at the boundary but leave costs littered throughout typed code. In either case, the performance penalty can become too high. If so, the developer faces a performance-debugging scenario.

One such performance-debugging scenario is found in the migration of the fsm program found in the GTP benchmark suite [28, 34]. The program is the creation of Nguyen and Andreozzi [54], economists interested in simulating an economy of agents with deterministic strategies. Figure 1a shows the outline of the four-module program: auto implements deterministic agents as state machines; pop coordinates among machines; main drives the simulation; and util provides helper functions.[3]

The configuration of fsm on the left of Figure 1a is fully untyped. If a developer adds deep types to the main module, performance is significantly degraded. The mixed-typed configuration suffers a 2.9x slowdown[4] compared to the untyped one. Switching to shallow types is a one-line change to the module, but does not remedy the situation; the

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

[4] This paper uses "slowdown" and "overhead" interchangeably, and both are expressed in terms of a multiplicative factor. If configuration A takes 1 second to run, and configuration B takes 2.9 seconds to run, then configuration B is said to have "2.9x slowdown" or "2.9x overhead" relative to configuration A.



(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
-------------------------------------------------
```

```
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
  (-> ....)
```

(b) Statistical profiler output for the configuration where main has deep types.

(c) Boundary profiler output for the same configuration.

Fig. 1. Profiling during type migration.

code still runs 2.8x slower than the untyped version. At this point, the question is how to recover the performance of the untyped configuration.[5] Each option results in different kind of costs.

- One option is to roll back the addition of types. This requires the developer essentially to throw away the effort and knowledge baked into those types and lose their hard-won type-soundness guarantees.
- A second option is to add (deep or shallow) types to an arbitrary module connected to main—following a "hunch" like developers sometimes do—but doing so can easily make things worse. For example, if one adds shallow types to the auto module, then performance degrades further (a 9x slowdown, to be precise).
- A third option is to add deep types to every module. This removes all the type boundaries from fsm, allowing it to run on par with the untyped configuration. It may in fact run faster than the untyped configuration, if adding types allows the compiler to apply type-driven optimizations. However, such a choice represents a heavy migration effort, which a developer who wishes simply 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 amount of effort and without discarding types.

[5] Using the untyped program as a baseline makes sense in this context because Typed Racket is a migratory language; that is, it tacks a type system onto a fundamentally untyped language. If the experiment was dealing with a static-first language as in the world of Grift [41], then using the fully-untyped program as a baseline might be inappropriate.

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 function applications; and
- a *boundary profiler*, which attributes the cost of types-as-contracts to specific module boundaries [4, 72].

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, which attributes costs to contracts directly. In contrast, the cost of shallow checks should be easier to find with the statistical profiler, since shallow checks add to the overall cost of a function in a way that the boundary profiler is not able to see.

The bottom half of Figure 1 shows the output of the statistical profiler and the boundary profiler for the top-right configuration of fsm 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 than those from the pop module. Equally unclear is whether the column labeled Total, which corresponds to the time spent executing a function call and every dependency of that call, is a better guide than the column labeled Self, which corresponds to the time spent in a function call excluding the time spent executing its dependencies. 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 prefix indicates proxies that wrap untyped functions used by typed components. The contracts are displayed from most costly to least costly, and all three of the most costly contracts are between main and pop. Since main is typed and pop is untyped, the hint is to add types to pop.

Adding types to pop does improve performance. Concretely, the resulting configuration suffers from a mere 1.2x slowdown. If this overhead is acceptable, the developer is done; otherwise, the hunt should 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 configurations 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 program: add types to an untyped component, or toggle a typed component from deep to shallow or vice versa. The next question is which component to modify. Since profiling tools identify parts of the program 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 faithfully follow 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 [36, 48]. 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 a 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* [69] 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.

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 stops if the performance is tolerable. 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 results. It then modifies the program based on this analysis and hands the modified version of the program 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 a fast configuration. Testing several strategies sheds light on their relative usefulness. If one strategy succeeds where another fails, it has higher value. Of course, the experiment may also reveal shortcomings of the profiling approach altogether—which would demand additional research from tool creators.

4 THE DESIGN OF THE RATIONAL-PROGRAMMER EXPERIMENT

Turning the sketch from the preceding section into a large-scale automated experiment requires rigorous descriptions for both the strategies and the notion of a performance-debugging scenario. As mentioned in the preceding section, given a scenario, a strategy identifies the next migration step, which should yield either a program with acceptable performance or another performance-debugging scenario. The preceding section also implies three possibilities for a migration step: (1) add types and specify their enforcement regime (deep, shallow); (2) toggle from one regime to another; or (3) abandon the search. Equipped with rigorous descriptions, the generic research question posed in the introduction becomes a set of quantitative questions.

Section 4.1 presents the strategies. Section 4.2 characterizes performance-debugging scenarios, which act as starting points for the migration process. Based on this description, it also lays out success and failure criteria. Finally, Section 4.3 formulates 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 types, and some are untyped. Independently of their types, a component c_1 may import another component c_2, which establishes between them a *boundary* across which they exchange values at run time. Depending on the existence and enforcement approach of types at each side of the boundary, a value exchange might trigger run-time checks, which degrade performance.

A rational-programmer strategy should thus aim to identify and eliminate the most costly checks in a program. In rigorous terms, a rational-programmer strategy is a function that consumes a program P and returns a set of pairs (c, t). Here c identifies a program component; 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 proposes a new variant of P where component c has deep types; if c is already typed, the strategy just requests toggling from shallow to deep. *Profiler-dependent* strategies use a profiler on P to generate the set of pairs, while *profiler-agnostic* strategies generate the set without using a profiler. An empty set indicates failure.

Basic strategies. Table 1 describes six *basic* strategies that rational programmers may use. These strategies are all *profiler-dependent*, but they differ at two levels: how to use profiler data to identify costly boundaries, and how to modify such boundaries.

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 procedure applications, each with two types of costs: the *total* time spent executing the call and its dependencies, and the *self* time spent in the call, which excludes dependencies. Because both costs are potentially useful, the rational programmers choose one of them. After ranking the applications by one of these two metrics, the rational programmers must identify a boundary. They start with the costliest-ranked application and seek a boundary between the enclosing component and a component with *stricter* types (since 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. If there are no applications remaining, the strategy returns the empty set.

At the second level, the *basic* strategies differ in how they migrate the two sides of their target boundary. Strategies that are *deep* turn the types on both sides of the boundary to deep. On the one hand, this action eliminates the cost of the boundary and enables type-driven optimizations in both components. On the other hand, it may also create costly boundaries to other components in a kind of ripple effect with potentially disastrous costs. By contrast, *shallow* strategies choose shallow types for both sides of the target boundary, unless the boundary is between a typed component and an untyped component, in which case the typed side becomes shallow and the untyped side is left unchanged. The

Profiler	Modification	Abbreviation	Description
boundary	*deep*	[BP.D]	Uses the boundary profiler to identify the most expensive boundary in the program. Recommends that both sides of the target boundary obtain deep types.
	shallow	[BP.S]	Uses the boundary profiler to identify the most expensive boundary in the program. Recommends that both sides of the target boundary obtain shallow types, unless the boundary is between an untyped module and a deep-typed module, in which case the typed side is changed to shallow, and the untyped side is left unchanged.
statistical(*self*)	*deep*	[SP.D]	Uses the statistical profiler to identify the component c_1 that contains the application with the highest *self* time in the program, and that has a boundary with at least one component c_2 that has stricter types than c_1. Recommends deep types for c_1 and c_2.
	shallow	[SP.S]	Like *statistical*(*self*) *deep*, with shallow types for c_1, c_2. If one of c_1, c_2 is typed and the other is deep-typed, then the deep-typed component is changed to be shallow-typed, and the untyped component is left unchanged.
statistical(*total*)	*deep*	[TP.D]	Like *statistical*(*self*) *deep* with *total* in place of *self*
	shallow	[TP.S]	Like *statistical*(*self*) *shallow* with *total* in place of *self*

Table 1. How the *basic* strategies find and modify slow boundaries.

rationale behind this choice is that changing a component from deep-typed to shallow-typed may lower the cost of that component's boundaries without the risk of a ripple effect.

☞ Greenman et al. [31] use the terminology of "optimistic" and "conservative" strategies. "Optimistic" strategies are the same as this paper's *deep* strategies. "Conservative" strategies differ from *shallow* strategies in the way they treat boundaries where one side is deep-typed and the other side is untyped. Specifically, "conservative" strategies turn both sides to shallow types, while *shallow* strategies turn only the deep side to shallow and leave the untyped side untouched.

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 than *profiler-dependent* strategies, then feedback from the profiler evidently plays a meaningful role. Otherwise, comparisons among *profiler-dependent* strategies are meaningless.

Table 2 lists the *profiler-agnostic* strategies used as baselines for the experiment. The first two, *random deep* and *random shallow*, aim to invalidate the null hypothesis with random choices. Specifically, *random deep* picks a random boundary with types of different strictness and updates the sides of the boundary in the same way as the *deep* strategies. *Random shallow* picks a random boundary with types of different strictness and updates the sides of the boundary in the same way as the *shallow* strategies. The second *profiler-agnostic* strategy, *toggling*, is due to Greenman [27] 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.

Strategy	Abbreviation	Description
random deep	Ran.D	Chooses a random boundary from the set of all boundaries where each side has a different type strictness. Recommends that both sides of the selected boundary obtain deep types.
random shallow	Ran.S	Chooses a random boundary from the set of all boundaries where each side has a different type strictness. Recommends that both sides of the selected boundary obtain shallow types.
toggling	Tog	Changes all typed modules to be deep-typed and tests the performance, then changes all typed modules to be shallow-typed and tests the performance. Chooses whichever one of these two configurations is better.

Table 2. The *profiler-agnostic* strategies used as baselines.

Profiler	Modification	Abbreviation	Description
boundary	*cost-aware deep*	$([BP.D])	Splits the boundaries in the program to those between typed components and the rest. Delegates to *boundary deep* to produce a modification for the program, but ranks boundaries in the first group higher than those in the second group.
	cost-aware shallow	$([BP.S])	Like *boundary cost-aware deep*, but delegates to *boundary shallow*.
	configuration-aware	#([BP.S]/[BP.D])	If less than 50 % of components in the program have types, delegates to *boundary shallow*. Otherwise, delegates to *boundary deep*.
statistical(*self*)	*cost-aware deep*	$([SP.D])	Delegates to *statistical*(*self*) *deep* to produce a modification for the given program, but prioritizes boundaries between typed components when determining which boundary to modify.
	cost-aware shallow	$([SP.S])	Like *statistical*(*self*) *cost-aware deep*, but delegates to *statistical*(*self*) *shallow*.
	configuration-aware	#([SP.S]/[SP.D])	Like *boundary configuration-aware*, but delegates to *statistical*(*self*).
statistical(*total*)	*cost-aware deep*	$([TP.D])	Like *statistical*(*self*) *cost-aware deep*, but delegates to *statistical*(*total*) *deep*.
	cost-aware shallow	$([TP.S])	Like *statistical*(*self*) *cost-aware deep*, but delegates to *statistical*(*total*) *shallow*.
	configuration-aware	#([TP.S]/[TP.D])	Like *boundary configuration-aware*, but delegates to *statistical*(*total*).

Table 3. The *type-aware* strategies, which use profiler data and current types to modify a boundary.

Type-aware 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 typed components in the program. With few types, the cost of introducing one component with deep types may be very high; with many types, the chance of a ripple effect is

Strategy	Abbreviation	Description
statistical(*self*) *flip*	F([BP.D]/[SP.D])	Like *boundary deep*, but the first time it gets stuck, it switches to *statistical*(*self*) *deep*. Here, stuck means that the profiler does not produce any actionable information. If *statistical*(*self*) *deep* gets stuck, the strategy switches back to *boundary deep*. And so on.
statistical(*total*) *flip*	F([BP.D]/[TP.D])	Like *boundary deep*, but the first time it gets stuck, it switches to *statistical*(*total*) *deep*. If *statistical*(*total*) *deep* gets stuck, the strategy switches back to *boundary deep*. And so on.
random flip	F([BP.D]/[Ran.D])	Like *boundary deep*, but the first time it gets stuck, it switches to *random deep*. *Random deep* never gets stuck, so the strategy continues to use *random deep* from that point onward.
statistical(*self*) *one-try*	1([BP.D]/[SP.D])	Like *boundary deep* but whenever it gets stuck, it uses *statistical*(*self*) *deep* to choose the next migration step. After that one choice, it switches back to *boundary deep*. This process repeats.
statistical(*total*) *one-try*	1([BP.D]/[TP.D])	Like *boundary deep* but whenever it gets stuck, it uses *statistical*(*total*) *deep* to choose the next migration step. After that one choice, it switches back to *boundary deep*. This process repeats.
random one-try	1([BP.D]/[Ran.D])	Like *boundary deep* but whenever it gets stuck, it uses *random deep* to choose the next migration step. After that one choice, it switches back to *boundary deep*. This process repeats.

Table 4. *Second-chance* strategies that change behavior when *boundary deep* fails.

probably low. Hence, the experiment includes strategies that take into account the types currently in the program before choosing how to respond to profiler data.

Table 3 lists these *type-aware* strategies, which are all *profiler-dependent*. 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 combine *deep* or *shallow* in an attempt to avoid ripple effects. When less than half of the components have types, they choose *shallow*, and when 50 % or more have types, they choose *deep*. The theory behind the *configuration-aware* strategies is that sparsely-typed configurations get more performance benefit shallow types than mostly- or fully-typed configurations, but this threshold of 50 % is arbitrarily chosen.

Second-chance strategies. The *profiler-dependent* strategies presented so far get stuck when the profiler does not provide any useful information. In this case, a developer may reach for another profiler. For example, the boundary profiler may fail to identify any boundaries as being costly or pin the cost on a boundary between a typed component of the program and an untyped library. Since the library is outside the developer's control, the information is not actionable. In these types of cases, the developer may choose to switch their strategy entirely, in hopes of navigating beyond their current dead end. For this reason, the experiment includes strategies that switch to a backup strategy when needed.

Table 4 lists these *second-chance* strategies, which switch between a primary and secondary strategy. The strategies form two groups based on when switches happen. Strategies in the first group, *flip*, attempt to use the secondary strategy for as many steps as possible after the primary strategy gets stuck, and return to the primary strategy only

when the secondary gets stuck. Strategies in the second group, *one-try*, resort to the secondary strategy to make the next step when the primary strategy gets stuck, and revert back to the primary strategy after that step.
☞ Greenman et al. [31] do not consider these *second-chance* strategies.

4.2 Navigation of migration lattices

This rational-programmer experiment demands a corpus of programs where every module is available in two flavors: typed and untyped. Since mechanically adding types to untyped code is an open and challenging problem[6], it makes sense to draw such a corpus from a benchmark suite where humans have already chosen type annotations. The established GTP benchmarks [28, 34] vary in size and complexity, embody a wide range of Racket programming styles, and come with well-chosen type annotations for all their components. Hence, the migration lattices can be pre-constructed for all benchmarks. It is thus possible to apply a strategy to any performance-debugging scenario in a program's migration lattice and use the strategy's recommendations to chart a path through that lattice.

Intuitively, a rational programmer using strategy S attempts to convert a program P_0 into a 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 its predecessor. The rational programmer thus constructs a *migration path*, a sequence of programs $P_0, \ldots, P_n$ inside a migration lattice. A migration path ends if a rational programmer's strategy cannot make a recommendation.

The Migration Lattice. All programs P_i are nodes in the *migration lattice* $\mathcal{L}[\![P_t]\!]$, where P_t is like P_i except that all of its components are deep-typed.[7] Each component in P_i might have deep types like P_t, shallow types, or no types at all. The bottom element of $\mathcal{L}[\![P_t]\!]$ is P_u, the untyped program. The lattice consists of *levels* of incomparable configurations. Every configuration at the same level has the same number of typed components. The 3^N nodes of $\mathcal{L}[\![P_t]\!]$ are ordered; $P_i \leqslant P_j$ if the untyped components in P_j form a subset of those in P_i.

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 rigorous equivalent to the description from the preceding section that strategies either add types to a single previously untyped component or toggle the type enforcement regime of existing typed components.[8] In other words, a migration path is a weakly ascending chain in $\mathcal{L}[\![P_t]\!]$.

Performance-debugging scenarios and success criteria. Completing the description of the experiment demands answers to two more questions. The first concerns the selection of starting points for the strategy-driven migrations, i.e., the *performance-debugging scenarios*. The question is which configurations qualify as slow. Let $perf(P_i)$ denote the running time of a program P_i. Since type checks are the source of performance overhead, the appropriate way to measure costs is to compare the performance of a configuration P to the performance of the untyped configuration P_u:

> *Given a migration lattice* $\mathcal{L}[\![P_t]\!]$*, a* performance-debugging scenario *is a configuration* P *such that* $perf(P) > T \cdot perf(P_u)$*, where* T *is the maximum acceptable performance degradation.*

[6] See An et al. [3], Campora et al. [9, 10], Castagna et al. [12], Chandra et al. [13], Cristiani and Thiemann [15], Furr et al. [20, 21], Garcia and Cimini [22], Jesse et al. [38], Kristensen and Møller [40], Malik et al. [46], Migeed and Palsberg [47], Miyazaki et al. [49], Phipps-Costin et al. [56], Rastogi et al. [57], Saftoiu [62], Siek and Vachharajani [68], Wei et al. [82], Yee and Guha [84]. For any untyped component, a migrating developer has to choose type annotations from among a potentially infinite number of possibilities. To make a rational-programmer experiment computationally feasible, it is necessary to avoid this dimension.

[7] There are many possible choices for the exact type annotations of P_t, and each denotes a unique lattice. By contrast, basing a lattice on an untyped program ($\mathcal{L}[\![P_u]\!]$) would be ambiguous without a pre-determined set of types.

[8] No strategy, including the profiler-agnostic ones, modifies a boundary where both sides are untyped.

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 $P_0 \dots P_n$ in a lattice $\mathcal{L}[\![P_t]\!]$ is strictly successful *iff*

(1) *P_0 is a performance-debugging scenario,*

(2) *$perf(P_n) \leq T \cdot perf(P_u)$, and*

(3) *for all $0 < i \leq n$, $perf(P_i) \leq perf(P_{i-1})$.*

To achieve strict success, following 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:

A migration path $P_0 \dots P_n$ in a lattice $\mathcal{L}[\![P_t]\!]$ is k-loosely successful *iff*

(1) *P_0 is a performance-debugging scenario,*

(2) *$perf(P_n) \leq T \cdot perf(P_u)$, and*

(3) *for all $0 < i \leq n$ with at most k exceptions, $perf(P_i) \leq \min\limits_{0 \leq j < i} perf(P_j)$*

equivalently: $|\{i \mid 0 < i \leq n : perf(P_i) > \min\limits_{0 \leq j < i} perf(P_j)\}| \leq k$.

The construction of a k-loose successful migration path allows for temporary performance degradations. The constant k is an upper bound on the number of degradations.

A patient developer may tolerate an unlimited number of setbacks:

A migration path $P_0 \dots P_n$ is N-loosely successful iff

(1) *P_0 is a performance-debugging scenario, and*

(2) *$perf(P_n) \leq T \cdot perf(P_u)$.*

4.3 The experimental questions

Equipped with rigorous definitions, it is possible to formulate the research questions precisely. The first two questions, Q_X and $Q_{X/Y}$, are the primary focus of this paper and concern the success rate of a strategy X, both in absolute terms (Q_X) and in comparison with another strategy Y ($Q_{X/Y}$). The third question, $Q_M([BP.D])$, zeroes in on a *cost-aware* strategy and its non-*cost-aware* counterpart as a preliminary exploration of how different strategies compare in terms of the programmer labor they require.

The first two experimental questions, which we explore for every strategy, are as follows:

Q_X How often does strategy X chart successful migration paths?

$Q_{X/Y}$ Does strategy X chart successful migration paths more often than strategy Y?

Answering Q_X boils down to determining, for every typed program P_t, the proportion of successes to failures of X across the performance-debugging scenarios in $\mathcal{L}[\![P_t]\!]$. Since lattice size varies significantly across different programs, and since bigger lattices tend to contain more performance-debugging scenarios, it is important to ensure that larger lattices do not have undue weight on the answer. Dividing each lattice's success count by its total number of performance-debugging scenarios and averaging this proportion across all lattices provides a better general picture of a a strategy's successfulness than summing up the raw success counts across all lattices does.

☞ Greenman et al. [31] aggregate successes and failures across all lattices rather than calculating the average success rate per lattice. This inevitably biases the answers by giving more weight to large lattices.

Using this average-based definition, a successful strategy X charts migration paths that are strictly successful for a high proportion of scenarios on average across all lattices. Essentially, a high proportion is evidence that when a rational programmer reacts to profiler feedback following X, it is likely to improve performance.

The above description uses the strict notion of success, which sets a high bar. To meet this standard, not only must the strategy guide the rational programmer to a configuration with a tolerable level of performance, but it must also bring the rational programmer monotonically closer to the target with each suggestion. Replacing the notion of strict success with 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 Q_X evaluates a strategy X in absolute terms, $Q_{X/Y}$ considers the value of X relative to some other strategy Y. This second question asks whether the average proportion of scenarios where X succeeds and Y fails is higher than the average proportion of scenarios where Y succeeds and X fails. Of course, the answer may not be clear-cut, as X and Y may do equally well or have complementary success records. In such a situation, it is illuminating to analyze whether X does better than Y in scenarios coming from different lattices, or whether they instead succeed and fail in the same cases.

When X is *profiler-dependent*, and Y is *profiler-agnostic*, and the answer to $Q_{X/Y}$ is positive, the experiment provides evidence against the null hypothesis. Such an answer suggests that X owes its success not to sheer luck or guesswork, but to information it gained from the profiler.

In sum, 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 program lattices.
(2) Calculate the migration paths for each strategy for each scenario in each lattice.
(3) Compare the successes and failures of the strategies.

Isn't Gradual Typing Dead? Although Takikawa et al. [74] show that many configurations of the GTP benchmarks run slowly, they do not answer Q_X and $Q_{X/Y}$—even in the N-loose case. Greenman et al. [34] attempt to investigate an N-loose version of Q_X in a 2^N lattice but severely limit the length of paths. Greenman [27] considers 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 significant slowdowns can be systematically transformed into 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.

Our third research question, $Q_M([BP.D])$, is an approximation and instantiation of the general question $Q_\$(X)$:

$Q_\$(X)$ Does the *cost-aware* version of strategy X require less programmer effort than its non-*cost-aware* equivalent?

This question is built on the idea that some updates are "easier" than others. For instance, switching a shallow-typed module to deep is easy because it only requires a user to change one line of code. Adding deep or shallow types to an untyped module, on the other hand, likely requires a large amount of effort. The programmer has to read or re-read the code, think about what types the code should have, and write in the type annotations. The idea behind the *cost-aware* strategies is that they prefer "easier" updates, which means that if the success rate of a *cost-aware* strategy is similar to its non-*cost-aware* sibling, then it is in theory preferable to that sibling because the developer does not have to do as much work when following it. $Q_\$(X)$ asks whether *cost-aware* strategies demonstrate this work-saving effect in practice.

To make $Q_{\$}(X)$ concretely answerable, we must choose a way to quantify the programmer effort of any given migration path. The question of how much effort a programmer must expend to add types to a given module is nebulous. One might use the number of lines of code in the module as a proxy, or perhaps the number of characters, unique identifiers, functions, higher-order functions, etc. To avoid this complexity, we assume that adding types to each module in a benchmark requires approximately the same amount of time and effort. The question then becomes:

$Q_{\boldsymbol{M}}(X)$ Does the *cost-aware* version of strategy X make programmers add type annotations to fewer **modules** than its non-*cost-aware* equivalent?

Attempting to summarize an answer to this question across 16 benchmark programs and 6 pairs of *cost-aware*/non-*cost-aware* strategies is a challenge. The number of modules to which it is possible to add types varies with how close the starting scenario lies to the top of the lattice, so one must be careful when comparing this metric across different benchmarks, or even across starting points in the same benchmark. A fair comparison between any two strategies requires an in-depth look stratified by benchmark and lattice level. Hence, to compare all 6 *cost-aware* strategies with their non-*cost-aware* counterparts is a sizeable task. Instead, we choose one such pair to compare in detail:

$Q_M([BP.D])$ Does the *boundary cost-aware deep* strategy make programmers add type annotations to fewer modules than the *boundary deep* strategy?

To answer this question, we will calculate the number of modules to which developers have to add types, averaged across all the paths in each group defined by the following characteristics: strategy (*boundary deep* vs *boundary cost-aware deep*), benchmark, lattice level, and success (i.e., whether the strategy achieves strict success or not). We will then look for trends in the comparison of each *boundary deep* number to its corresponding *boundary cost-aware deep* number.

5 RESULTS

Running the rational-programmer experiment requires a large pool of computing resources. To begin, the experimental setup demands reliable measurements for all complete migration lattices. Then, the rational programmers need 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). Sections 5.2 and 5.3 explain how the outcome of the experiment answers the two research questions from the preceding section. Section 5.5 presents a close look at the results for the best-performing strategy, and Section 5.6 looks at results for alternative values of the maximum acceptable performance degradation T. Section 5.7 examines results when paths' starting points are filtered according to various criteria.

5.1 Experiment

The experiment uses the v7.0 release[9] of the GTP Benchmarks with some small restructurings to help the boundary profiler attribute costs correctly. These restructurings do 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;[10] gregor,quadT, and quadU because each has over

[9] https://github.com/utahplt/gtp-benchmarks/tree/v7.0

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

Table 5. Details of data-collection runs.

Data collected	Server	Racket	Typed Racket
runtimes and profiles (dungeon)	c220g2	v8.6.0.2 [cs]	29ea3c10
runtimes and profiles (morsecode)	m510	v8.6.0.2 [cs]	700506ca (cherry pick)
runtimes (all other benchmarks)	c220g1	v8.6.0.2 [cs]	default
profiles (all other benchmarks)	m510	v8.6.0.2 [cs]	default

Table 6. Server details.

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

1.5 million configurations, 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.

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 5 shows the division of work across servers from CloudLab [16]. Each server ran a sequence of measurement tasks and nothing else; no other users ran jobs during the experiment's reservation time. Table 6 gives 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 [19]) 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.[11] Fixing these issues was not necessary for the rational-programmer experiment per se, but makes the outcome more relevant to current versions of Racket.

Dealing with sampling error. In the mathematical world of Section 4.2, comparing the running times of two configurations P_1 and P_2 is straightforward, since every configuration can be thought of as having exactly one runtime; $perf(P_1) < perf(P_2)$ if and only if the runtime associated with P_1 is a smaller number than the runtime associated with P_2. But when there are multiple samples associated with P_1 and multiple samples associated with P_2, the truth value of $perf(P_1) < perf(P_2)$ becomes potentially ambiguous, especially if there is overlap between the ranges of the two sample sets. To clarify this ambiguity, our experiment relies on a statistical-significance-based definition of <. We define $perf(P_1) < perf(P_2)$ to be true for a given pair of samples if and only if (A) the sample mean for P_1 is less than

[11] See https://github.com/racket/typed-racket/pull/1282 and https://github.com/racket/typed-racket/pull/1316

Table 7. Sample statistics for absolute runtimes, in milliseconds, of each benchmark program's untyped configuration (8 samples each).

Benchmark	Sample Min	Sample Mean	Sample Max	Sample Std. Dev.
morsecode	2679	2727.2	2800	44.52
forth	15	15.1	16	0.33
fsm	324	325.6	327	1.11
fsmoo	599	604.5	622	6.98
mbta	1338	1361.2	1430	27.62
zombie	65	68.0	70	1.87
dungeon	127	130.6	136	2.91
jpeg	304	311.2	319	5.97
lnm	524	538.5	552	10.28
suffixtree	3589	3642.1	3700	34.75
kcfa	1063	1091.9	1174	32.72
snake	680	688.4	699	6.02
take5	674	703.0	737	17.99
acquire	417	427.2	437	7.22
tetris	665	671.5	681	4.30
synth	406	416.4	433	8.25

the sample mean for P_2, and (B) a Welch's t-test finds with 95 % confidence that there is a significant difference between the two population means.

A similar test is applied to determine the truth value of $perf(P_1) \leq T \cdot perf(P_2)$ for some constant T. The statement is considered to be true if and only if (A) P_1's sample mean is less than or equal to the result of multiplying P_2's sample mean by T, or (B) a 95 %-confidence Welch's t-test does not find there to be a significant difference between the two population means. Importantly, the t-test does not take the constant T into account at all. Doing so is unnecessary because in every version of our experiment, $T \geq 1$, and as T grows, the t-test becomes less and less important, since (A) becomes true for more and more pairs of samples. Thus, there is no need to modify the t-test for alternate values of T; the t-test adjudicates the edge cases when T is close to 1 and fades in importance when T is larger.

This redefinition of < and ≤ necessitates another look at the usage of min in the definition of k-loose success in Section 4.2. Since ≤ is no longer a partial order, min in this context is ill-defined. The solution is to replace $\min_{0 \leq j < i} perf(P_j)$ with $\textit{best-perf}(i-1)$, where $\textit{best-perf}(j) = \begin{cases} perf(P_j) & \text{if } j = 0 \text{ or } perf(P_j) \leq \textit{best-perf}(j-1) \\ \textit{best-perf}(j-1) & \text{otherwise.} \end{cases}$
This definition clarifies the order by which configurations are tested to be the best-performing-so-far. It corresponds to an algorithm that starts with the first configuration in the path and scans the path in order, replacing the best-so-far configuration P_b with the configuration currently being scanned P_c whenever $P_c \leq P_b$.

☞ Greenman et al. [31] use a different method to judge whether $perf(P_1) < perf(P_2)$ should be considered true. In their setup, $perf(P_1) < perf(P_2)$ if and only if $m_1 + s_1 < m_2 - s_2$, where m_1 and s_1 are the sample mean and sample standard deviation of P_1, and m_2 and s_2 are the sample mean and sample standard deviation of P_2. The method this paper uses is more closely aligned with standard statistical practices.

Basic Observations. Table 7 displays statistics summarizing the recorded runtime data for each benchmark's fully-untyped configuration. Table 8 shows the number of modules in each benchmark, the number of configurations

Table 8. The percentage of the 3^N configurations that are performance-debugging scenarios.

Benchmark	N	3^N	% Scenario
morsecode	4	81	86.42 %
forth	4	81	95.06 %
fsm	4	81	77.78 %
fsmoo	4	81	83.95 %
mbta	4	81	88.89 %
zombie	4	81	91.36 %
dungeon	5	243	99.59 %
jpeg	5	243	94.65 %
lnm	6	729	41.56 %
suffixtree	6	729	98.49 %
kcfa	7	2,187	93.51 %
snake	8	6,561	99.98 %
take5	8	6,561	99.95 %
acquire	9	19,683	99.34 %
tetris	9	19,683	96.17 %
synth	10	59,049	99.99 %

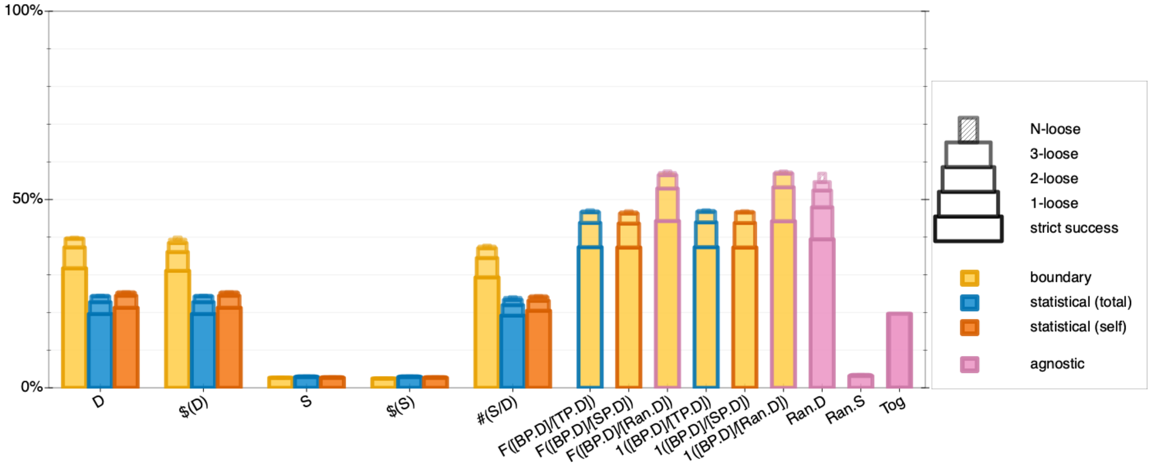


Fig. 2. Average success rates for $T = 1$. Yellow represents the boundary profiler, while blue and orange represent the statistical profiler. Pink represents the lack of a profiler.

implied by that number of modules, and the percentage of these configurations with bad enough performance to be considered performance-debugging scenarios. These measurements confirm that the GTP benchmarks are suitable for the rational-programmer experiment. With $T = 1$ as the goal of migration (T, introduced in Section 4.2, is the maximum acceptable performance degradation), all but two benchmarks have plenty of performance-debugging scenarios. Going by configurations rather than benchmarks, more than 98 % of all configurations are interesting starting points for the experiment.

5.2 Answering Q_X

Figure 2 presents the success rates of the 23 strategies from Section 4.1, averaged across the 16 GTP benchmark programs listed in Table 8. It answers research question Q_X from Section 4.3.

Each stacked bar in Figure 2 corresponds to a different strategy. Concretely, the bars report a strategy's success rate for increasingly loose notions of success when $T = 1$. The lowest, widest part of each bar represents the percentage of scenarios where the strategy achieves strict success. The next three levels represent 1-loose, 2-loose, and 3-loose success. The thinnest bar is for N-loose successes. The strategies result in a wide range of success rates:

Table 9. How many scenarios can possibly reach 1x without removing types?

Benchmark	# Scenario	% Hopeful	Benchmark	# Scenario	% Hopeful
morsecode	70	100.00 %	lnm	303	100.00 %
forth	77	6.49 %	suffixtree	718	100.00 %
fsm	63	100.00 %	kcfa	2,045	100.00 %
fsmoo	68	100.00 %	snake	6,560	0.00 %
mbta	72	0.00 %	take5	6,558	0.00 %
zombie	74	35.14 %	acquire	19,553	4.44 %
dungeon	242	0.00 %	tetris	18,930	100.00 %
jpeg	230	100.00 %	synth	59,046	100.00 %

- *Deep* (“D”) modification performs moderately well when guided by the *boundary* profiler, finding strict success slightly over 30 % of the time. With a 2-loose relaxation, success rises to around 40 %. The results are worse, however, with *statistical*(*total*) or *statistical*(*self*) profiling, both of which stay within the 20-25 % range.
- *Cost-aware deep* (“$(D)”) is almost as successful as *deep* when driven by *boundary* and equally successful with *statistical*(*total*) and *statistical*(*self*).
- *Shallow* (“S”) modification is successful in under 5 % of cases, no matter which profiler guides it.
- *Cost-aware shallow* (“$(S)”) modification is just as unsuccessful as *shallow*.
- *Configuration-aware* (“#(S/D)”) has a success rate 2-3 % lower than *deep*’s across the board.
- All of the *flip* (“F(· · ·)”) strategies perform better than the *basic*, *cost-aware*, and *configuration-aware* strategies. In particular, the *random flip* (“F([BP.D]/[Ran.D])”) strategy fares the best in this group, with a success rate above 40 % for strict success, and above 50 % for 2-loose success. (The success rates for *random flip* are averaged across 10 trials. The standard deviation of each percentage was low: under 0.4 % for each success rate.)
- Each of the *one-try* (“1(· · ·)”) strategies perform similarly well when compared to its analogous *flip* strategy. *Random one-try* (“1([BP.D]/[Ran.D])”) is once more the best of the group. (The success rates for *random one-try* are calculated as for *random flip*, and just as for *random flip*, the standard deviation of each rate calculated was under 0.4 %.)
- The *random deep* (“Ran.D”) strategy does quite well on its own, achieving nearly 40 % success without any profiler. It achieves 2-loose success over 50 % of the time. The *random shallow* (“Ran.S”) strategy does just as poorly as the other *shallow* strategies, achieving success in less than 5 % of cases. (As with *random flip* and *random one-try*, these results are the average success rates across 10 trials, and have standard deviations under 0.4 %.)
- *Toggling* (“Tog”) achieves strict success about 20 % of the time. The other notions of success do not apply to *toggling* because it stops after one step.

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 9 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, snake, and take5, which have zero hopeful scenarios. Three others are only marginally better: forth, zombie, and acquire.

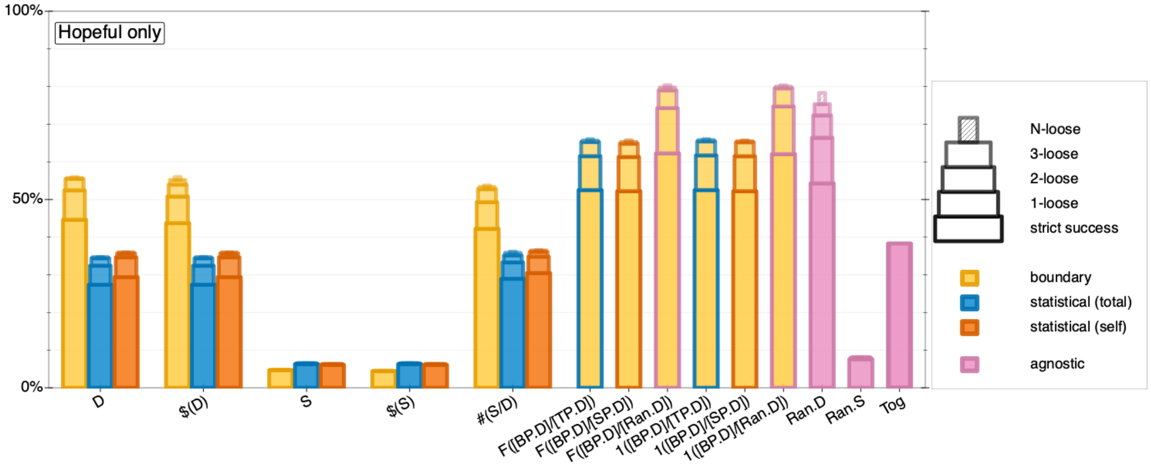


Fig. 3. Average success rates for $T = 1$, hopeful only.

Figure 3 therefore revisits the measurements reported in Figure 2, focusing on hopeful benchmarks and scenarios only. If there is no possible migration path from a performance-debugging scenario to a configuration with a tolerable overhead, the scenario is excluded as hopeless. As before, the results for *random flip*, *random one-try*, *random deep*, and *random shallow* are the average across 10 runs. The standard deviations of each metric are higher than before, but remain under 0.8 % for all three of these strategies.

Every strategy performs better when zeroing in on these hopeful scenarios, but all of the trends from Figure 2 remain the same. The *second-chance* strategies still do the best, with *random flip* and *random one-try* achieving strict success rates over 60 % and a 3-loose success rates close to 80 %. The *shallow* and *cost-aware shallow* strategies still do the worst, with N-loose success less than 10 % of the time.

5.3 Answering $Q_{X/Y}$

The preceding subsection hints at how the strategies compare to each other. *random one-try* and *random flip* are the most likely to succeed on an arbitrary performance-debugging scenario. The rest of the *second-chance* strategies are less likely to succeed. *Deep*, *cost-aware deep*, and *configuration-aware* strategies follow. The *shallow* strategies are least likely to find a successful configuration no matter what profiler they use.

Between *random one-try* and *random flip*, the former is preferable for a couple of reasons. First, each success rate for *random one-try* has a lower standard deviation than *random flip* across 10 trials. This means that, while the two strategies have an approximately equal chance of success on average, *random one-try*'s success rate is more dependable than *random flip*'s. Additionally, *random one-try* resorts to random chance less often. Unlike *random flip*, it avoids asking developers to make arbitrary choices unless no other option is available. For these reasons, we designate *random one-try* to be the *recommended* strategy.

The question remains of whether there are particular cases in which the other strategies succeed and *random one-try* fails. To address it, Figure 4 compares the *random one-try* strategy to all others. It thus answers question $Q_{X/Y}$. The x-axis lists all 23 strategies, including *random one-try* itself (fourth from the right, labeled as 1([BP.D]/[Ran.D])). The y-axis reports average proportions of scenarios per lattice. For each strategy, there are at most two vertical bars. The

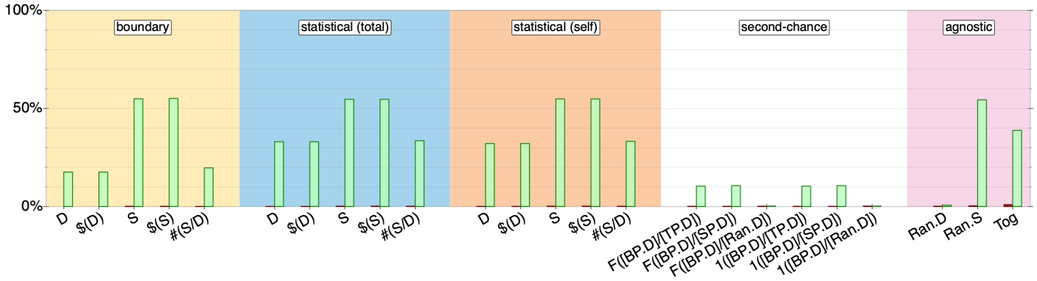


Fig. 4. *Random one-try* vs. the rest, strict success: losses (red bars) and wins (green bars) on all scenarios.

height of a green bar denotes the average proportion of scenarios per lattice where *random one-try* succeeds and the other fails. The height of a red bar reports the reverse; that is, the average proportion of scenarios per lattice where *random one-try* fails and the other strategy succeeds. Ties do not count; hence, the red and green bars do not combine to 100 %.

The tiny red bars and tall green bars give a negative answer to the question of whether *random one-try* performs worse in a significant number of cases. Other strategies rarely succeed where *random one-try* fails. The strategy that fares the best on this front is *toggling*, but the number of scenarios where *random one-try* succeeds and *toggling* fails dwarfs the number of scenarios where *toggling* succeeds and *random one-try* fails. Thus, *random one-try* is still much more successful than *toggling*, even though there are a small number of cases where *toggling* finds success and *random one-try* does not.

5.4 Answering $Q_M([BP.D])$

Tables 10 and 11 compare the average (mean and median, respectively) number of modules for which a developer is forced to come up with types when using the *boundary deep* and *boundary cost-aware deep* strategies. This is defined as the number of modules that are untyped at the beginning of the path, and have shallow or deep types at the end of the path. The "end" of the path in this context corresponds to the point in the path when either (1) success is reached, or (2) the current point in the path has a greater runtime than the previous point in the path, breaking the criteria the strict success. When the two numbers (rounded to the nearest tenth of a module) are the same, only one is shown, and when the two numbers are different, the *cost-aware* version is shown in parentheses below the non-*cost-aware* number. Whenever no path exists for a given benchmark-level-successfulness combination, the table displays a dash (–) instead of a number.

Table 10 shows a slight trend towards saved labor when one uses the *cost-aware* version of the strategy. Of the 155 benchmark-level-successfulness combinations that yield paths, 21 display a lower mean for *cost-aware*, and only one (suffixtree, level 5, success) displays a higher mean. Granted, the rounded difference is no greater than 0.1 for any of these pairs, so this table does not provide a clear answer to the question of whether using *boundary cost-aware deep* saves the developer a significant amount of labor compared to *boundary deep*, in terms of either statistical or practical significance.

Since the median tends to correspond to a "typical case", Table 11 provides a more practically applicable answer to the question. In the entirety of this table, only one slot displays a difference between *boundary deep* and *boundary cost-aware deep*, suggesting that typically, a developer will have to add types to the same number of modules, regardless of

Table 10. On average, how many modules does a programmer have to add type annotations to when navigating with the *boundary deep* and *boundary cost-aware deep* strategies? Green boxes report averages for strictly successful paths, while red boxes report averages for failure paths.

Benchmark	Mean num. modules (*cost-aware*) *boundary deep* adds types to before strict success/failure																			
morsecode	1.0	0.5	1.2	0.5	0.8	0.2	0.0	0.0												
forth	–	1.1	–	0.7	–	0.4	–	0.0												
fsm	2.0	0.5	1.1	0.4	0.6	0.2	0.0	0.0												
fsmoo	–	1.1	1.2	0.8	0.8	0.2	0.0	0.0												
mbta	–	1.2	–	0.9	–	0.5	–	0.0												
zombie	–	0.6	1.0	0.6	0.0	0.5	–	0.0												
dungeon	–	0.6	–	0.8	–	0.7	–	0.5	–	0.0										
jpeg	1.6	0.9	1.4	0.9	1.2	0.7	0.7	0.4	0.0	0.0										
lnm	1.0	0.0	1.0	0.2	1.0	0.2	0.7	0.2	0.4	0.1	0.0	0.0								
suffixtree	3.0	1.7	2.8	1.8	2.2	1.5	1.5 (1.6)	1.1 (1.0)	0.8	0.6	0.0	0.0								
kcfa	2.3	1.2	2.2 (2.1)	0.9	1.8	1.0	1.4	0.9	1.1	0.6	0.6	0.2	0.0	0.0						
snake	–	2.0	–	2.4 (2.3)	–	2.4 (2.3)	–	2.2	–	1.8	–	1.3	–	0.7	–	0.0				
take5	–	1.2	–	1.6	–	1.7 (1.6)	–	1.6 (1.5)	–	1.4 (1.3)	–	1.0 (0.9)	–	0.6 (0.5)	–	0.0				
acquire	–	1.0	1.0	1.3	1.0	1.4 (1.3)	0.0	1.4	–	1.3	–	1.1	–	0.8	–	0.4	–	0.0		
tetris	3.5	2.9	3.0 (2.9)	2.5	2.6	2.3	2.3 (2.2)	2.0	1.8	1.7	1.4 (1.3)	1.3	0.9	0.9 (0.8)	0.5 (0.4)	0.4	0.0	0.0		
synth	–	2.4	8.0	3.2	7.0	3.4 (3.3)	6.0	3.2 (3.1)	5.0	2.8 (2.7)	4.0	2.3 (2.2)	3.0	1.7 (1.6)	2.0	1.1 (1.0)	1.0	0.5	0.0	0.0
Level	1		2		3		4		5		6		7		8		9		10	

🟩 *boundary deep* success	🟥 *boundary deep* failure
(*cost-aware* *boundary deep* success)	(*cost-aware* *boundary deep* failure)

which version of the strategy they use. Since *boundary cost-aware deep* already has a worse success rate (Figure 2), there is clearly no advantage to using the *cost-aware* version.

5.5 A closer look at *random one-try*

Given that *random one-try* is the recommended strategy, it is worth taking a closer look at how it operates when navigating each benchmark's lattice. To that end, this section answers three questions about *random one-try*: how often it resorts to its secondary strategy; how likely it is that it reaches success given the proportion of the lattice that is

Table 11. How many modules does a programmer typically have to add type annotations to when navigating with the *boundary deep* and *boundary cost-aware deep* strategies?

Benchmark	Median num. modules (*cost-aware*) *boundary deep* adds types to before strict success/failure																			
morsecode	1	0	1	0	1	0	0	0												
forth	–	1	–	1	–	0	–	0												
fsm	2	1	1	0	1	0	0	0												
fsmoo	–	1	1	1	1	0	0	0												
mbta	–	1	–	1	–	0	–	0												
zombie	–	1	1	1	0	0	–	0												
dungeon	–	0.5	–	1	–	1	–	0	–	0										
jpeg	2	1	1	1	1	1	1	0	0	0										
lnm	1	0	1	0	1	0	1	0	0	0	0	0								
suffixtree	3	1	3	2	2	2	2	1	1	1	0	0								
kcfa	2	1	2	1	2	1	1	1	1	1	1	0	0	0						
snake	–	2	–	2	–	2	–	2	–	2	–	1	–	1	–	0				
take5	–	1	–	1	–	1	–	1	–	1	–	1	–	1	–	0				
acquire	–	1	1	1	1	1	0	1	–	1	–	1	–	1	–	0	–	0		
tetris	3.5	4	3	3	3	2	2	2	2	2	1	1	1	1	0	0	0	0		
synth	–	1	8	2	7	4	6	4	5	3	4	3	3	2	2	1	1	1 (0)	0	0
Level	1		2		3		4		5		6		7		8		9		10	

🟩 *boundary deep* success	🟥 *boundary deep* failure
(*cost-aware boundary deep* success)	(*cost-aware boundary deep* failure)

already typed at the beginning of the navigation; and how many type additions are needed when navigating with this strategy.

To answer the question of how often *random one-try* resorts to its secondary strategy, Figure 5 displays the average number of steps that it takes across all strictly successful paths, broken down by its *boundary deep* and *random deep* sub-strategies. Since *random one-try* switches back and forth between *boundary deep* and *random deep*, the average number of *random deep* steps approximates the number of times *boundary deep* gets stuck along a typical strictly successful *random one-try* path. Hence, comparing that number with the average number of *boundary deep* steps answers whether *random one-try* relies heavily on its secondary *random deep* strategy. For most benchmarks, the answer is negative; *random one-try* does not rely heavily on *random deep*. However, there are benchmarks where the opposite is true. At one extreme is suffixtree, kcfa, and tetris, which almost never resort to *random deep*, and at the

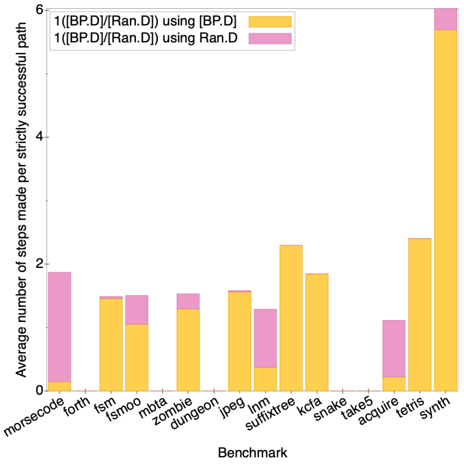


Fig. 5. Average number of steps taken with *boundary deep* (yellow) vs. *random deep* (pink) during successful *random one-try* navigations.

other is morsecode, lnm, and acquire, which do so a majority of the time. In this first set of benchmarks, the basic *boundary deep* strategy outperforms the basic *random deep* strategy, while in the second set, the basic *random deep* strategy outperforms (see Appendix B). This suggests that *random one-try*'s success comes from relying heavily on the *random deep* strategy in situations where *random deep* performs better, and relying heavily on *boundary deep* in situations where *boundary deep* performs better. That is, it offers the best of both worlds.

To shed light on the likelihood of success from a given starting point, Table 12 shows the average number of strictly successful and failed paths that *random one-try* follows when it starts navigation from a given lattice level. (Levels are numbered according to the number of typed components. Level 0 contains the wholly-untyped configuration, level 1 contains all configurations where exactly 1 component is typed, and so on.) For some benchmarks, such as zombie, there is a greater chance of success when starting from a lower point in the lattice, while for others, such as synth, starting from a higher level makes success more likely. For a third group, which includes suffixtree, the chance of success peaks around the middle of the lattice. The benchmarks forth, mbta, dungeon, snake, and take5 comprise a fourth group for which *random one-try* never finds success. Overall, the chance of success or failure seems to be program-specific, and cannot be estimated solely from the number of typed modules in the program.

To get a sense for the amount of type information a developer will have to add when navigating with *random one-try*, Table 13 gives the median number of modules to which types must be added when *random one-try* starts from a given lattice level. For strictly successful paths, this number is often smaller than the distance to the top of the lattice, which shows that *random one-try* does not chart only paths that go all the way to a fully-typed configuration. Rather, it typically finds performant configurations that are nearby. The most glaring exceptions are fsmoo and synth, where strictly successful paths typically require the developer to climb to the very top of the lattice. (Table 14, which displays the number of configurations that have an overhead of 1x or better, makes it clear why this is the case for synth. Outside of the very bottom of its lattice, synth has only one configuration with acceptable performance, and it is at the highest lattice level. In the case of fsmoo, it is unclear why this occurs.) Table 13 also shows that, just as for successful paths, *random one-try*'s failure paths do not typically go all the way to the top of the lattice. Here, the exceptions are dungeon,

Table 12. How likely is strict success with the *random one-try* strategy (on average, across 10 trials)?

Benchmark	Likelihood of strict success, *random one-try*									
morsecode	0.45	0.69	0.81	0.67						
forth	0	0	0	0						
fsm	0.65	0.85	0.96	1						
fsmoo	0.65	0.77	0.73	0.68						
mbta	0	0	0	0						
zombie	0.20	0.39	0.16	0						
dungeon	0	0	0	0	0					
jpeg	0.58	0.55	0.67	0.74	0.71					
lnm	1	0.96	0.96	0.96	0.98	0.95				
suffixtree	0.46	0.65	0.77	0.82	0.81	0.74				
kcfa	0.65	0.75	0.74	0.75	0.78	0.78	0.80			
snake	0	0	0	0	0	0	0	0		
take5	0	0	0	0	0	0	0	0		
acquire	0.03	0.02	0.01	0	0	0	0	0	0	
tetris	0.75	0.49	0.48	0.53	0.59	0.66	0.73	0.80	0.85	
synth	0.16	0.28	0.40	0.50	0.58	0.65	0.72	0.76	0.78	0.76
Level	1	2	3	4	5	6	7	8	9	10

Table 13. How many modules must be typed during a *random one-try* navigation (median across the scenarios on each lattice level and across 10 trials)?

🟩 *random one-try* success 🟥 *random one-try* failure

Benchmark	Median num. modules *random one-try* adds types to before strict success/failure																			
morsecode	1	2	2	1	1	1	0	0												
forth	–	2	–	2	–	1	–	0												
fsm	2	1	1	1	1	0	0	–												
fsmoo	3	1	2	1	1	1	0	0												
mbta	–	2	–	1	–	1	–	0												
zombie	2	1	1	2	0	1	–	0												
dungeon	–	1	–	3	–	2	–	1	–	0										
jpeg	2	1	2	1	1	1	1	1	0	0										
lnm	1	–	1	1	1	1	1	1	0	0	0	0								
suffixtree	3	1	3	1	2	1	1	1	1	0	0	0								
kcfa	2	1	2	1	2	1	1	1	1	1	0	0	0	0						
snake	–	1	–	3	–	5	–	4	–	3	–	2	–	1	–	0				
take5	–	1	–	1	–	1	–	4	–	3	–	2	–	1	–	0				
acquire	1.5	1	1	2	1	2	0	2	0	2	–	2	–	2	–	1	–	0		
tetris	5	1	3	1	3	1	2	1	2	1	1	1	1	0	0	0	0	0		
synth	9	1	8	1	7	1	6	1	5	1	4	1	3	1	2	0	1	0	0	0
Level	1		2		3		4		5		6		7		8		9		10	

snake, and take5, which are all hopeless benchmarks. In sum, the *random one-try* strategy usually does not require the addition of types to all modules of a program, but there are cases where this does end up happening.

Table 14. Which levels of the migration lattice have any acceptable configurations?

Benchmark	# acceptable				
morsecode	2	3	3	2	
forth	2	1	0	0	
fsm	2	4	7	4	
fsmoo	2	4	2	4	
mbta	4	4	0	0	
zombie	2	3	1	0	
dungeon	0	0	0	0	0
jpeg	2	1	1	4	4
Level	1	2	3	4	5

Benchmark	# acceptable									
lnm	9	37	90	137	113	39				
suffixtree	1	0	0	1	4	4				
kcfa	8	24	32	20	16	26	15			
snake	0	0	0	0	0	0	0	0		
take5	2	0	0	0	0	0	0	0		
acquire	7	27	42	39	13	1	0	0	0	
tetris	12	49	112	139	107	90	106	100	37	
synth	1	0	0	0	0	0	0	0	0	1
Level	1	2	3	4	5	6	7	8	9	10

5.6 Relaxing the maximum acceptable performance degradation

The presentation of the results so far consider a rather stringent maximum acceptable performance degradation, $T = 1$. While parity with the performance of untyped code is an ideal goal for mixed-typed programs built on dynamic languages, developers may be willing to trade some run-time costs for labor, and they may have to make the best of a situation if the ideal goal proves unreachable. This section revisits the experimental question Q_X from Section 5.2 under this light.

Figures 6 to 9 present results for alternative T-values. Figure 6 shows the strategies' average success rates across all benchmarks when $T = 1.1$. It is generated by the exact same process as Figure 2, except that a configuration must have at least 1.1x slowdown compared to the untyped configuration to qualify as a performance-debugging scenario, and a migration path must end in at most 1.1x slowdown compared to the untyped configuration to qualify as successful. Figure 7 shows the hopeful-only success rates across all benchmarks—unlike when $T = 1$, every benchmark has some hopeful scenarios under $T = 1.1$. Figures 8 and 9 show the regular and hopeful-only success rates for $T = 1.5$ across 14 benchmarks. They exclude lnm because this benchmark no longer offers any performance-debugging scenarios under $T = 1.5$.

Naturally, when standards are relaxed, standards are more often met. Overall, the general trends from $T = 1$ are preserved, even as the success rates of all strategies increase. Specifically, *random one-try* and the rest of the *second-chance* strategies remain the top performers, and the *shallow* and *cost-aware shallow* strategies still do significantly worse than the rest. The most striking change from the data in Section 5.2 is that the effect of focusing on hopeful scenarios becomes muted. While the success rates of the strategies for $T = 1$ increase significantly when considering only hopeful scenarios, the differences are less pronounced for $T = 1.1$ and quite faint for $T = 1.5$. Understandably, an increase in T reduces the size of hopeless regions in the lattices of most benchmarks, which causes most benchmarks' scenario sets to match their hopeful-only scenario sets more closely, which in turn causes strategy success results to change less when only hopeful scenarios are considered. An exception to the above observation is the *toggling* strategy, which jumps quite a bit when the $T = 1.1$ results zoom in on hopeful scenarios.

5.7 Grouping by starting scenario

Although a performance-debugging developer does not know the performance characteristics of their program's lattice as a whole, they do know where their current scenario lies in the program lattice and everything about the migration path that has brought them there. If different strategies have different relative success rates depending on the

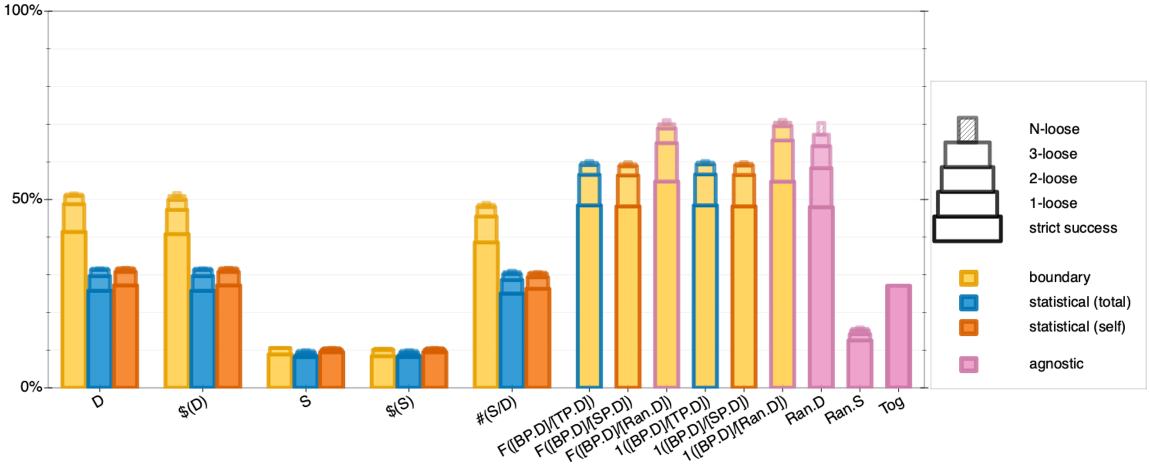


Fig. 6. Average success rates for $T = 1.1$.

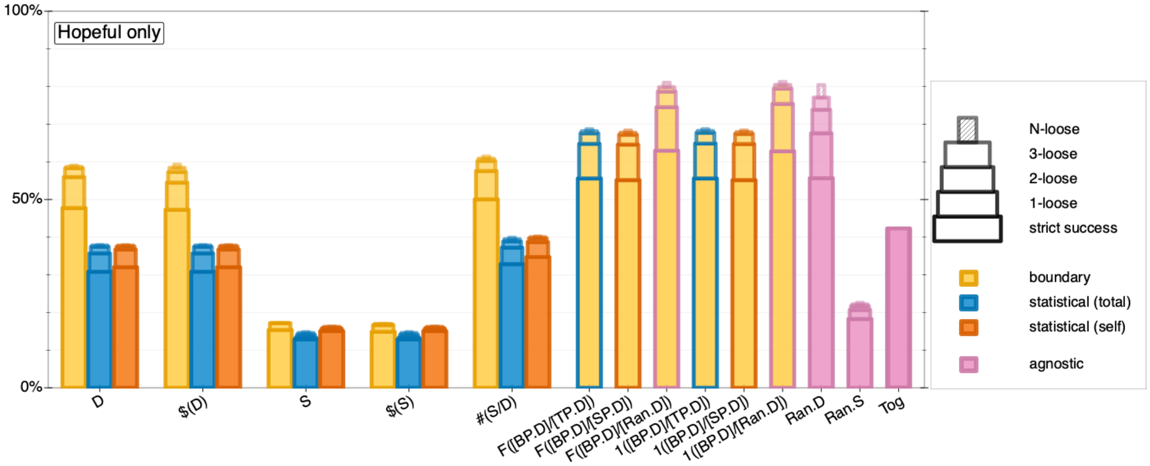


Fig. 7. Average success rates for $T = 1.1$, hopeful only.

characteristics of the path's initial starting point, then practical insight can be gained by exploring how the results look when they are grouped by various starting-scenario criteria. This section takes a look at the results filtered by two such factors: lattice height and the "deepness" or "shallowness" of the initial scenario.

Lattice height is an important factor to consider because the unbalanced structure of the program lattices may impact the success rate of various strategies. To see this, suppose a program has N modules. Then the top layer of the lattice has 2^N configurations, the layer below that has $2^{N-1}N$ configurations, the layer below that has $2^{N-2}(N(N-1)/2)$ configurations, and so on. Meanwhile, the bottom layer of the lattice has 1 configuration, the layer above that has $2N$ configurations, the layer above that has $4(N(N-1)/2)$ configurations, and so on. Clearly, the top half of the lattice, which covers all of the scenarios where typed modules outnumber untyped modules, drowns out the bottom half of the lattice by at least an order of magnitude. The bottom levels of the lattice yield far fewer paths than the top levels

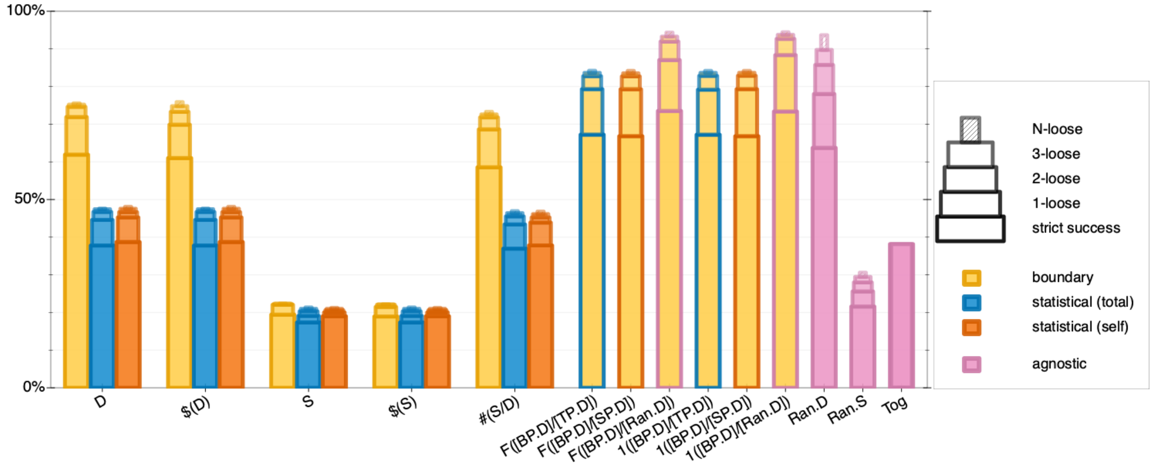


Fig. 8. Average success rates for $T = 1.5$.

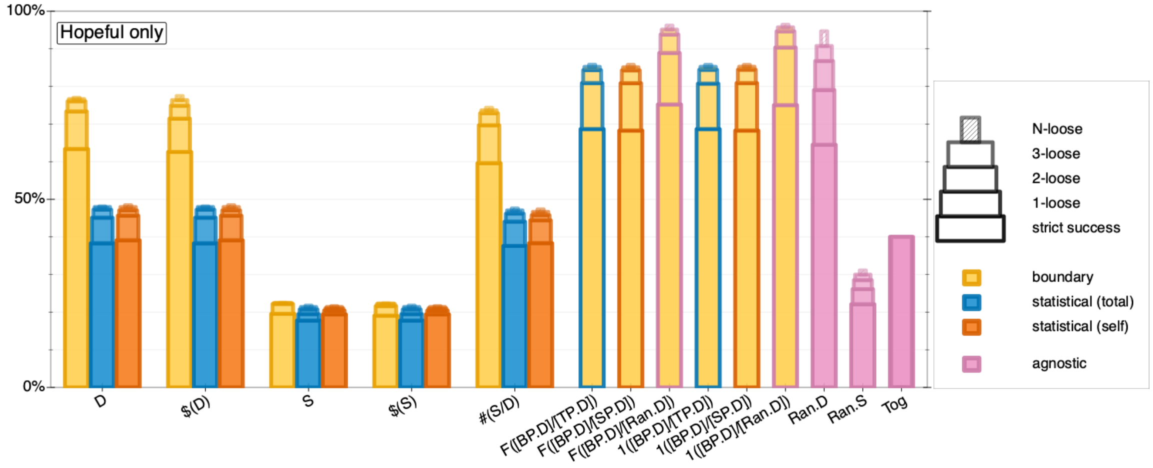


Fig. 9. Average success rates for $T = 1.5$, hopeful only.

because the bottom levels yield far fewer performance-debugging scenarios. Since the results as presented so far give equal weight to each performance-debugging scenario in a benchmark, they are naturally biased toward navigations that start higher up in the lattice. This may disadvantage *shallow* strategies, which, according to folklore, should work better for configurations with few typed modules.

To evaluate the severity of this drowning-out effect, Figure 10 filters the results such that the set of performance-debugging scenarios is limited to those located in the bottom half of the lattice. For benchmarks where the lattice has an odd number of layers, scenarios in the middle layer are not considered to be part of the bottom half. These bottom-half results show generally similar trends to the full-lattice charts. The *configuration-aware* strategies are an exception, which is unsurprising because *configuration-aware* strategies behave differently depending on the lattice level. Recall that the *configuration-aware* strategies have *shallow* behavior in the bottom half of the lattice and *deep*

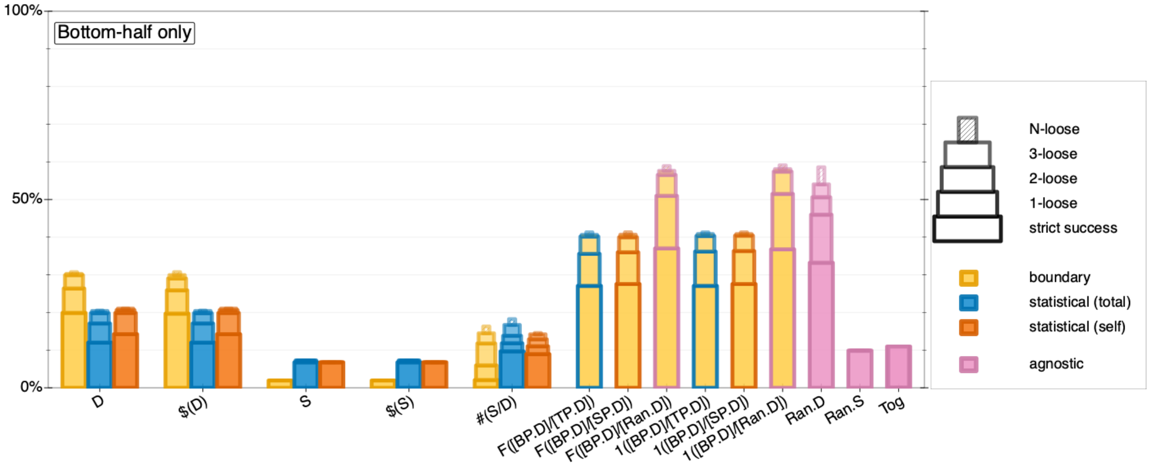


Fig. 10. Average success rate across for $T = 1$, lattice bottom-half only.

behavior in the top half. When starting from lower levels of the lattice, the *configuration-aware* strategies generally find success one of two ways: by having their *shallow* strategy take them to a nearby acceptable configuration, or by having their *shallow* strategy stumble around until they reach the top half of the lattice and *deep* takes over. This results in a strict success rate similar to *shallow* and non-strict success rates better than *shallow*. Another difference present in the bottom-half results is, in fact, the success rates of the *shallow* strategies. Strategies that lean *shallow* get a small bump when not paired with the *boundary* profiler, but they are still the worst-performing overall. For instance, the vanilla *statistical*(*total*) *shallow* strategy jumps from around 4 % success to 8 % success when zooming in on the bottom-half scenarios, but the *deep*-leaning strategies remain more successful. Even when looking only at paths that start from the very first lattice level (Figure 11), *shallow* remains worse than *deep*. Overall, since the general trends stay the same when focusing on lower levels of the lattice, the natural bias towards higher-level scenarios likely does not affect the validity of the results as presented in Section 5.2.

Another interesting factor to consider is how the shallowness or deepness of the initial performance-debugging scenario affects a strategy's chance of success. In order to begin exploring this question, Figure 12 shows the results when paths that include deep types in their initial performance-debugging scenario are excluded. Similarly, Figure 13 shows the results when paths whose starting configuration includes shallow types are excluded. For both of these charts, the successfulness of each strategy relative to the others remains largely the same, with the notable exception of *toggling*, which is extremely unsuccessful. Apparently, if all of a developer's typed modules use the same kind of checks, and there is a performance problem, then flipping all the typed code to use a different kind of check is not helpful. This makes a certain sense, as *toggling* does not alter the boundaries between typed and untyped code at all, and performance-debugging scenarios arise from cross-boundary interaction. It seems that *toggling* only ever finds success by eliminating boundaries between deep and shallow, not by managing to reduce the cost of interactions between typed and untyped code.

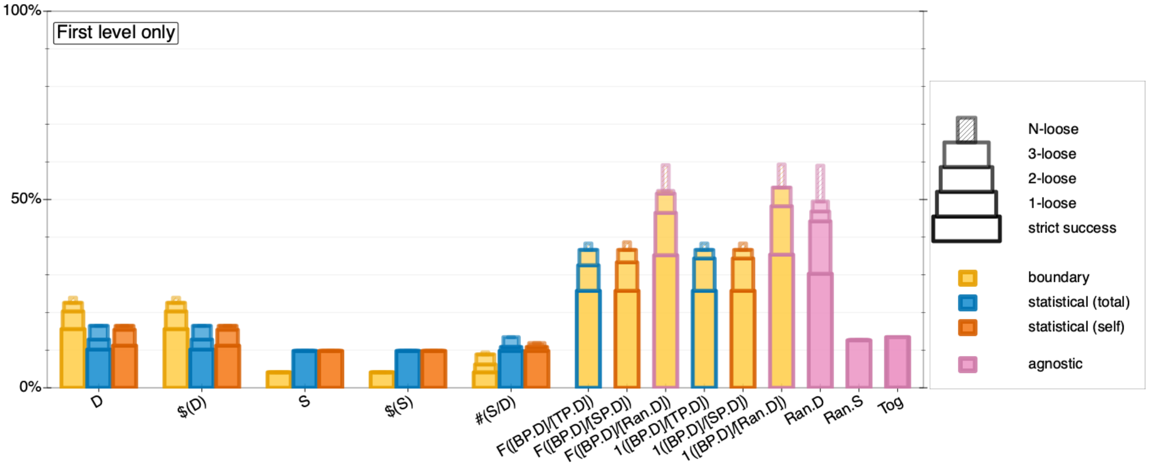


Fig. 11. Average success rates for $T = 1$, starting from first lattice level (one typed module) only.

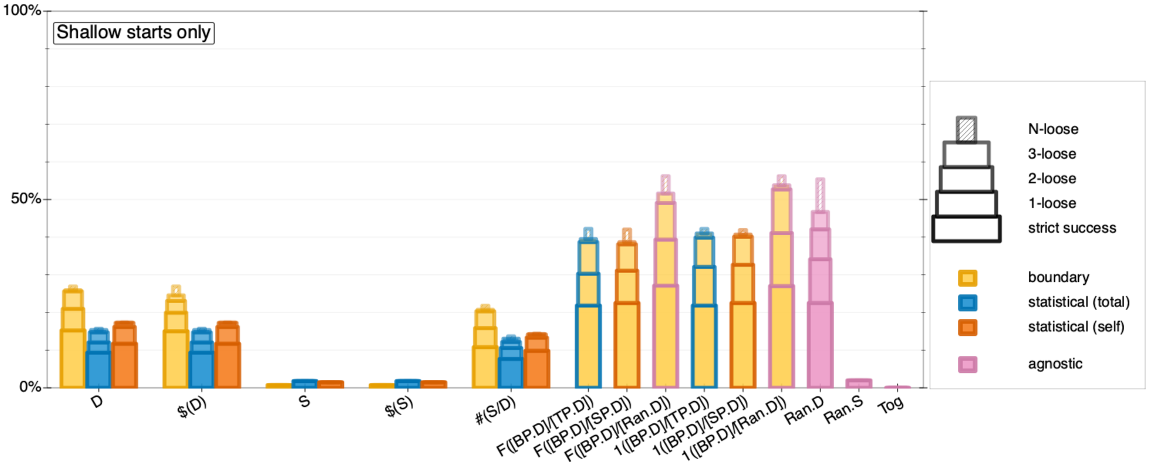


Fig. 12. Average success rates for $T = 1$, shallow starting points only.

6 LESSONS FOR DEVELOPERS AND LANGUAGE DESIGNERS

The results of the rational-programmer experiment suggest a few concrete lessons for developers (Section 6.1) and language designers (Section 6.2). Of course, readers should be aware of a few threats to the validity of the data and conclusions (Section 6.3).

6.1 Lessons for developers

When a developer faces a performance-debugging scenario, the first two questions are whether to reach for a profiling tool and if so, what kind. The general results and the results for many individual benchmarks[12] suggest an answer to both of these questions. In all benchmarks besides forth and acquire, where *toggling* is marginally better than the rest

[12] Appendix B provides a detailed account of the experimental results for each benchmark individually.

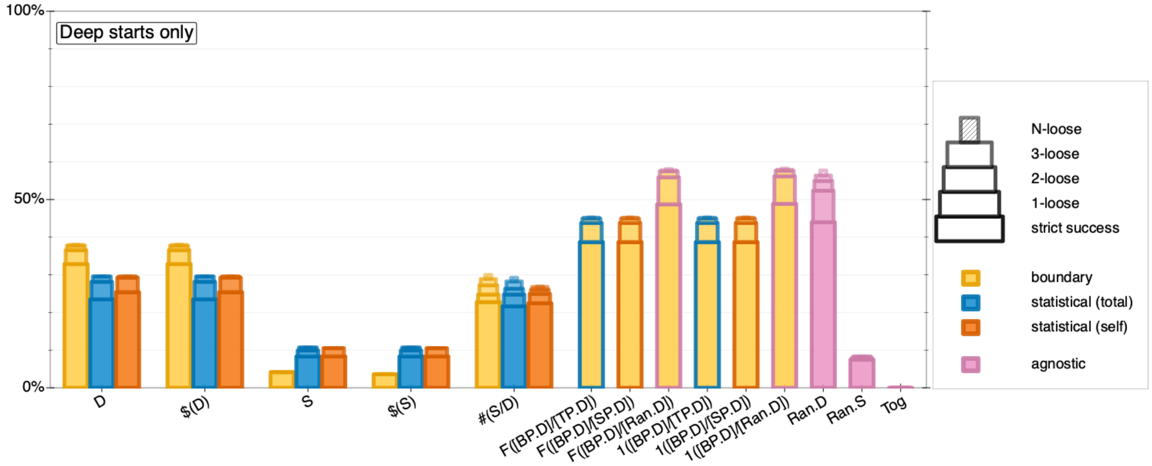
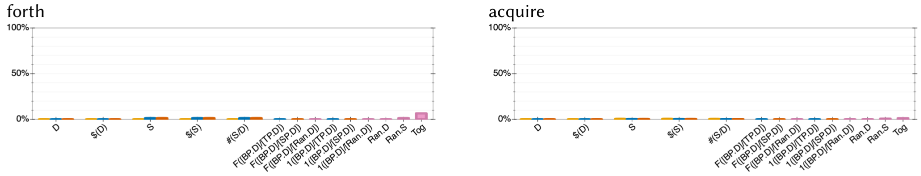


Fig. 13. Average success rates for $T = 1$, deep starting points only.


forth

acquire

Fig. 14. Programs with large hopeless regions.

(Figure 14), there is some profiler-based strategy that outperforms the *profiler-agnostic* strategies, and this winner is usually based primarily on the *boundary* profiler. Hence, when faced with a performance bottleneck due to runtime type checks, a developer should use a profiler, and that profiler should be a *boundary* one. Although the developer may find success using a *statistical* profiler if no boundary profiler is available, they are far more likely to succeed with a boundary profiler.

When a developer has reached for a profiler, the next question is how to interpret its output. The data suggest 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.

Since profilers often rely on statistical approximation, situations may occur where the boundary profiler produces uninformative or non-actionable output. In such a situation, a developer can give up, switch profilers, or ignore profiler feedback altogether. The basic *boundary deep* strategy represents this first option, while the *second-chance* strategies explore the other two options. Figure 4 shows that the *random one-try* strategy never fails on a scenario where the *boundary deep* strategy succeeds. Thus, giving up never does better than continuing onward. *Second-chance* strategies that switch to *random deep* tend to do better than those that switch to a *statistical* profiler, so when the *boundary deep* profiler fails, a developer's best option (within the space of strategies this paper explores) is to randomly choose a boundary where each side has a different type strictness, without the help of a profiler.

A reader may also wonder whether developers should give up once they reach a configuration that has worse performance than its predecessor. Figure 2 gives an ambivalent answer to this question. On the one hand, strategies based on the *boundary* profiler, including the *second-chance* strategies, exhibit a 3-loose success rate that is around 10 percentage points higher than for strict success. On the other hand, after one performance setback, it is highly unlikely that continuing the navigation will result in performance benefits, and there is a risk of degrading performance further. To illustrate, the *second-chance random* strategies fail over 75 % of the time once a single setback has occurred, even under N-loose success criteria.

Finally, a practical question is whether developers can extract value from profiling even when migration costs cannot be eliminated entirely. Figures 6 and 8 indicate that the answer is positive. Although no strategy gets above a 60 % success rate for eliminating overhead completely, the *second-chance random* strategies get runtime down to 1.5x that of the untyped configuration over 90 % of the time.

6.2 Lessons for language designers

The addition of shallow types to the implementation of Typed Racket [27] does not seem to help developers navigating the migration lattice in search of lower performance costs. The *shallow* navigation strategies are inferior to the *deep* navigation strategies in all benchmarks aside from the pathologically hopeless acquire. Perhaps existing profiling technology is simply not sophisticated enough to determine the aggregate cost of shallow type-assertions. Further research on techniques for compiling and profiling gradually typed languages seems necessary to unlock the potential of shallow type-enforcement, if any exists.

Even when shallow types are not in play, the experimental results reveal significant room for profiling improvements. For instance, the morsecode benchmark has 4 modules, and when all of them are untyped, the code takes between 2600 and 2800 milliseconds to run (across 8 trials). When the first 3 modules are given deep types and the last module remains untyped, it takes between 4400 and 5000 milliseconds to run (across 8 trials). Clearly, deep type enforcement for this second configuration adds around 2000 milliseconds to the run time. However, when the *boundary* profiler runs on this configuration, it returns no actionable information about where the cost is coming from, claiming that 0 % of the running time is taken up by deep type checks. Most likely, this results from the *boundary* profiler having too small a sampling frequency to ever sample during the execution of a contract check. A possible improvement to the *boundary* profiler would be the addition of a mode which counts the frequency of contract checks directly, rather than relying on regular sampling.

In general, both the *boundary* and *statistical* profilers fail to produce actionable information in many configurations with high overheads. With the *boundary* profiler, these non-actionable configurations exhibit a 2.6x overhead on average, and the most common reason for the lack of actionable information is that there are no internal boundaries in the profiler's output. This is either because there are no expensive deep type boundaries (i.e., the costs come mainly from the enforcement of shallow types) or because the boundaries involve at least one component that lives "outside" the program in library code. This no-internal issue affects 4051 configurations in the benchmarks. For these configurations, using the *statistical* profiler's guidance to take the next step might help the migration get unstuck. The success rates of the *second-chance* strategies that resort to *statistical*(*self*) *deep* and *statistical*(*total*) *deep* confirm that this is the case, up to a point. While these strategies do better than the basic *boundary deep*, they do worse than the *second-chance* strategies that resort to *random deep*. The likely reason is that the *statistical* profiler runs into the same type of issue as the *boundary* profiler. The *statistical* profiler's output contains no actionable boundaries for a sizable number of configurations (2347), even though these configurations exhibit sky-high slowdowns—on average 87x. There are several possible reasons:

the boundaries might point to library code; the identified costs might result from essential computations rather than gradual typing checks; or both sides of the identified boundary might be already typed.[13]

Overall, the lesson from these Typed Racket experiments is that designers of mixed-typed languages should invest in deep types and a profiler that can track their enforcement costs, rather than shallow types.

6.3 Threats to validity

The validity of the conclusions may suffer from two distinct kinds of threats. The first kind, method-internal, concerns the experimental setup. The second category is about extrinsic aspects of the rational-programmer method.

The most important method-internal threat is that the GTP Benchmarks may be unrepresentative of Racket programs in the wild. Several benchmarks are somewhat small with simple dependency graphs and low performance overheads. Since developers typically confront performance-debugging scenarios from large, high-overhead programs, they must apply the general lessons with some caution. The problem is that such programs may naturally tend to have large hopeless regions in the migration lattice. If a program configuration belongs to a hopeless region, then no strategy can possibly get it down to performance parity with the untyped configuration, and developers may have to settle with some performance degradation. It remains an open question how often hopeless regions occur in the wild. Additionally, the benchmarks used in the experiment are by no means a statistically representative sample of all Racket programs in the world—it is unclear how one would even go about taking such a sample. That said, the selection of benchmarks attempts to provide a diverse set of real-world Racket programs, covering a variety of authors, programming styles, and purposes. Greenman [28] extensively discusses the nature of the benchmarks; the interested reader may wish to study the descriptions therein.

The second method-internal threat concerns the design of the strategies, which exhibit three potential weaknesses. First, while the set of strategies covers basic approaches to navigation and strategies that combine some of these basic approaches, there are many combinations left unexplored. Specifically, only three strategy pairings are explored in total: *boundary deep* with *statistical*(*total*) *deep*, *boundary deep* with *statistical*(*self*) *deep*, and *boundary deep* with *random deep*.[14] An unexplored pairing of strategies, such as *statistical*(*self*) *deep* with *random deep*, may fare better than any of these, especially in situations like morsecode and lnm where the *boundary* profiler underperforms. A second potential weakness of the strategies is that they all, aside from *toggling*, take migration steps that modify exactly one or two modules at a time. Under any notion of success besides N-loose, even a so-called "hopeful" scenario may have no valid path to success that does not involve modifying three or more modules at once. Hence, a non-*toggling* strategy that can take steps modifying more than two modules at a time may do better than the current strategies. Alternatively, a strategy that operates at a granularity smaller than module-level may also do better. Such a strategy might find success by splitting a thorny module into several submodules and choosing to change the types of some submodules while keeping others the same. The third potential weakness is that strategies working with statistical profiler output find the maximum cost of any application in a given module and ignore the cost of other applications in the module.

[13] Interestingly, the statistical profiler seems to do better on programs where the maximum overhead is small. fsm, lnm, and kcfa are the only benchmarks where the *statistical deep* strategies have a success rate above 50 % (see Appendix B), and they exhibit relatively low overheads across all their configurations compared to the other benchmarks. While 10 of the 16 benchmarks have worst-performing configurations in the double, triple, or quintuple digits (forth has a configuration with 16185x overhead), the worst-case overheads of these three benchmarks are single-digit (9.39x for fsm, 1.22x for lnm, and 4.65x for kcfa).

[14] The reason we examine only three strategy combinations is that there are far too many possible pairings of basic strategies to explore them all. With 16 basic strategies, there are 240 possible pairs of a primary and secondary strategy. We chose *boundary deep* as the primary strategy for each pair based on the results from Greenman et al. [31], on the intuition that *boundary deep* was the best starting point and that the other profiler options might be able to provide actionable profiler information (or fake such information, in the case of *random*) in parts of the lattice where the *boundary* profiler cannot.

Some other scheme, perhaps one that involves adding up the cost of all applications in a module, may lead to better recommendations.

The third method-internal threat to validity is the effect of the GTP benchmarks' so-called "adaptor modules" on the performance of configurations. First introduced in Takikawa et al. [74], adaptor modules canonicalize type definitions, which are generative in Typed Racket. In essence, they prevent a situation where two typed modules ascribe different types to the same untyped module. Unfortunately, these adaptor modules always have deep types, even when none of the other modules in a benchmark are deep. Therefore, despite the fact that the types defined in the adaptor modules of the GTP benchmarks are rather simple, their deep quality can affect the performance impact of the runtime type checks that take place when shallow-typed modules interact with untyped ones. This is the case in fsm, lnm, and tetris, where turning untyped modules to shallow (a normally irrational action) can occasionally improve performance. The *shallow* strategy's results for these three benchmarks in Appendix F demonstrate this. The following benchmark programs use adaptor modules: acquire, fsm, fsmoo, kcfa, lnm, snake, suffixtree, synth, take5, tetris, and zombie. The remaining benchmarks (dungeon, forth, jpeg, mbta, and morsecode) do not. While the full extent of the adaptor modules' impact on the performance of the benchmarks is unknown, the results of this paper's experiments do not display a notable difference in trends between these two groups (for details, see Appendix B).

A final internal threat is that the large scale of the experiment imposes feasibility constraints on the collected data. Specifically, the experiment collects (and averages) only eight runtime measurements per scenario, and runs each profiler only once per scenario. This data was collected on single-user machines, and the output did not seem to present any major instabilities. Nevertheless, the reader should keep this and all of the method-internal threats listed so far in mind when drawing conclusions.

As for the method-external threats, the most important one is that the experiment relies on a single language (Typed Racket) and its associated ecosystem. This choice is necessary for two different reasons. First, from a methodological perspective, an apples-to-apples comparison of strategies demands an experimental environment that controls for their differences. Second, from a feasibility perspective, Racket is the only existing production language that (i) is equipped with a statistical and a boundary profiler and (ii) enables programmers to mix deep and shallow types in a single program. It is unclear how well this paper's results apply to other production languages built on different design choices than Racket, especially if those languages manage to successfully leverage optimizations for reducing run-time type-checking costs [11, 41, 80]. For instance, a recent experiment with Reticulated Python suggests that changes in the precision of type annotations can have varied and unpredictable impacts on the performance of a program [39]. Accordingly, it would be a good idea to re-investigate the effectiveness of this paper's navigation strategies in the context of an optimized production language that allows for a finer notion of migration than module-by-module. Although the authors cannot be certain of how exactly our conclusions would change in such a setting, we conjecture that hopeless scenarios would have a far milder impact, since ripple effects resulting from unrelated components being forced to share each other's type enforcement regime would go away completely. Another aspect of this threat is that the experiment involves only two profilers. It is possible that other language ecosystems offer profiling tools with different strengths and weaknesses than the two flavors investigated here.

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. Nevertheless, programming language researchers know quite well that simplified models have an illuminating power. Empirical PL research has relied on highly simplified mental models of program execution for a long time. As Mytkowicz et al. [53] 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 that of human developers.

That said, the simplifications of the rational-programmer method 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, the rational-programmer method produces concrete results that shed light on the pragmatic aspects of, in this case, debugging a mixed-typed program's performance overhead, and it does so at scale and quantitatively.

7 RELATED WORK

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 difficulties that come from enforcing the types of mixed-typed programs. Two significant sources of inspiration are research on gradual type migration and research on profiling techniques. At the meta-level, this work builds on and extends prior results on the rational-programmer method.

Performance of Mixed-Typed Programs. Greenman et al. [34] demonstrate the grim performance problems of deep 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 establishes the 3^N migration lattices that are key for the rational-programmer experiment herein and provides the blueprint for 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. [25, 32, 45, 58, 59, 61, 64, 65, 73, 77]). One notable strategy is transient, which was developed for Reticulated Python [78–81], adapted to Grace and JIT-compiled to greatly reduce costs [23, 61] and later characterized as providing *shallow* types that offer type soundness but not complete monitoring [30]. Greenman et al. [29] provide a detailed analysis and comparison of the overall checking strategy landscape.

The second group of mitigations reduces the time and space required by deep types without changing their semantics [2, 5, 6, 17, 37, 41, 50, 63, 67]. This is a promising line of work. In the context of Pycket, for example, the type migration problem is easier than it is in Typed Racket because many more configurations run efficiently. But even then, pathological cases remain. Good navigation techniques continue to be important, even as overall performance improves.

Several language designs adopt a hybrid type enforcement mechanism to avoid the costs of deep checks. Dart 1[15] and Gradualtalk [1] allow developers to manually toggle between deep and *optional* types. Optional types never introduce run-time checks (same as TypeScript [7] or Flow [14]). Following a more sophisticated approach, Thorn and StrongScript use a mixture of optional and *concrete* types [60, 83]. Concrete types perform cheap nominal type checks but limit the

[15] https://dart.dev/resources/language/evolution

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.[16] Nom [51, 52] and SafeTS [58] independently propose concrete types as a path to efficient gradual types. Static Python combines concrete and shallow types to ease the limitations of concrete [45]. Pyret uses deep checks for fixed-size data and shallow checks for recursive data and functions.[17] Typed Racket recently added shallow and optional types as alternatives to deep [27].

Gradual Type Migration. Research on gradual type migration can be split in three broad directions: static techniques [9, 12, 13, 21, 22, 40, 47, 56, 57, 68]; dynamic techniques [3, 15, 20, 49, 62], and techniques based on machine learning (ML) [38, 46, 82, 84]. The dynamic and ML-based techniques exhibit the most scalable results so far and can produce accurate annotations for a range of components in the wild, such as JavaScript libraries. However, as Yee and Guha [84] make clear, 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 rational-programmer strategies in this paper with migration techniques in the context of automatic or human-in-the-loop tools.

Herder [10] combines variational typing, a static migration technique, with a cost semantics in order to estimate relative performance. 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 (with, e.g., union and universal types), and further experiments are needed to test whether its approximations can find not just the best-case configuration but nearby satisficing ones.

Performance Tuning with Profilers. Profilers are the go-to tool for developers hoping to understand the causes of performance bugs. Tools such as GNU gprof [24] established a legacy of statistical (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 [72]. A feature-specific profiler groups execution time based on instances of language features of the developer's choosing, rather than by function calls. For instance, the experiment of this paper uses it as a boundary profiler, aggregating the cost of contracts in a program by the boundary that introduces them.

Two prior works have used profiler feedback to understand the source of mixed-typed programs' performance problems. First, Andersen et al. [4] show that the Racket feature-specific profiler can detect hot boundaries in programs that use deep types, i.e., that it can determine which boundaries are the origin of the costliest deep checks. Second, Gariano et al. [23] 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 mixed-typed programs does not examine how to translate profiler feedback into developer actions. In general, most profiling tools do not make recommendations to developers. The Zoom profiler [55] 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 of how to collect, organize, and display execution costs back to the developer. Two recent approaches developed for Java are of particular interest: vertical profiling [35] and concept-based profiling [70]. 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

[16] https://dart.dev/language/type-system#runtime-checks

[17] http://www.pyret.org

based on user-defined portions of a program called concepts [8]. It may be interesting to study alternative profiling and performance analysis techniques in future rational-programmer experiments.

The Rational Programmer. Lazarek et al. [42, 43, 44] propose and use the rational programmer as an empirical method for evaluating the role that blame assignment plays when locating errors in programs with software contracts and in gradually-typed programs. However, the core ideas of the rational programmer apply beyond locating logical errors. In essence, the rational programmer is a general methodological framework for investigating of the pragmatics of programming languages and tools systematically. That is, it can quantify the value of many different aspects of a language or tool in the context of many different tasks. In that sense, prior work focuses on a single context: debugging errors.

This paper shows how the rational programmer applies to experiment design in another context: performance debugging. Although the language feature it studies, gradual typing, is shared with prior work, this paper looks at a different aspect of the feature's pragmatics. As a result, it not only contributes to the understanding of gradual types but also provides evidence for the generalizability of the rational-programmer method itself.

8 ONWARD!

Mixed typing come with several advantages [42, 43] but poses a serious performance-debugging challenge to developers who wish to use it. Profiling tools are designed to help developers 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 21 competing strategies that each depend on one or both of two off-the-shelf profilers.

At the object level, the results deliver several insights:

(1) A developer's best bet is to use the *boundary* profiler with a *deep* strategy. That is, a developer hoping to bring down the cost of a mixed-typed program should use the boundary profiler to determine the hottest boundary and add deeply-enforced types to both sides. If no boundary profiler is available, the next-best thing is to use a *statistical* profiler with the same strategy.

(2) When a profiler fails provide actionable information, the developer should try to jump-start the rest of the migration by taking a *random deep* step.

(3) If, in following a strategy, the developer takes a migration step that slows the program down instead of speeding it up, it may be in their best interest to continue with the strategy, as tolerating a few setbacks is sometimes necessary to reach a configuration with satisficing performance. However, each setback reveals this happy ending to be less and less likely; after two misleading suggestions, the developer should switch strategies or give up.

(4) For certain kinds of programs, the migration lattice contains a huge region of hopeless scenarios in which no strategy can guide the program to performance parity with untyped code. These regions call for fundamental improvements to sound gradual typing.

(5) That said, these hopeless scenarios are not some inescapable pit where developers cannot benefit from profilers at all. Even when the total elimination of performance overhead is impossible, profilers can typically help developers get costs down to a manageable level (i.e., to an overhead under 1.5x compared to the all-untyped configuration).

(6) Finally, the results challenge Greenman [27]'s report that adding shallow type enforcement is helpful. While that work shows that toggling allows migration with a maximum 3x overhead in many scenarios, the poor results of (i) toggling and (ii) the *configuration-aware* strategies in this paper indicate that shallow checks are not a useful stepping stone towards mixed-typed migration with acceptable costs. If performance close to that of the 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 using rational programmers deliver results that contradict anecdotal reports from human developers. Of course, a rational programmer is not a human developer. It remains an open question how well the results apply to actual performance-debugging scenarios when human beings are involved.

Finally, the rational-programmer experiment suggests three main threads for future research. First, the experiment should be reproduced for other mixed-typed languages. Doing so would help determine whether the value of the boundary profiler and of the strategies that lean towards deep types is Racket-specific or a more general property of mixed-typed languages. Given the apparent fruitlessness of shallow types, one would likely gain useful information by replicating the parts of the experiment that do not deal with shallow types in languages that only have deep and optional type enforcement. As mentioned in Section 6.3, it would also be interesting to see how the results change when the language does not require type migration to have module-level granularity.

A second direction for further investigation would be to develop a method for predicting whether a given scenario is hopeless. In hopeless scenarios, a developer often has to add type annotations to multiple modules before they reach failure, representing a significant waste of time if they only care about getting performance down to 1x overhead. Being able to guess well whether the scenario is hopeless ahead of time would save a lot of time and effort. Possible inputs to this prediction method might include the current number of typed modules, the initial overhead of the scenario, or the ratio of the performance of the all-shallow version of the scenario to the all-deep version.

Third, the experiment demonstrates the need for better profiling tools. Although boundary profilers can help in many situations, there are often patches where the profilers fail to offer guidance, and the developer's best bet is to take a blind, random step. One potential starting point for investigation is to look for patterns determining what kinds of choices are more likely to lead to success within these blind patches of the lattices. As it stands, the existing profiling tools are not enough to smoothly navigate the performance challenges of mixed-typed migration.

Data availability statement. The data for this paper is available on Zenodo, along with scripts for reproducing the experiment and analyzing the results: https://doi.org/10.5281/zenodo.8136116.

ACKNOWLEDGMENTS

Hejduk and Dimoulas were partly supported by NSF grants SHF 2237984 and SHF 2412400. Originally, Felleisen and Greenman were partly supported by NSF grant SHF 1763922. Felleisen also received partial support via grants SHF 2007686, 2116372, and 2315884. Greenman received support from NSF grant 2030859 to the CRA for the CIFellows project. 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.

REFERENCES

[1] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2013. Gradual Typing for Smalltalk. *Science of Computer Programming* 96, 1 (2013), 52–69.

[2] Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-Typed Objects. In *DLS*. 27–36.

[3] 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

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

[5] 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

[6] 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.

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

[8] 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

[9] 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

[10] 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

[11] John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. 2024. Type-Based Gradual Typing Performance Optimization. *PACMPL* 8, POPL (2024), 89:1–89:33.

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

[13] 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

[14] 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.

[15] 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

[16] 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

[17] 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.

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

[19] 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

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

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

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

[23] 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.

[24] 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

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

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

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

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

[29] Ben Greenman, Christos Dimoulas, and Matthias Felleisen. 2023. 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

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

[31] Ben Greenman, Matthias Felleisen, and Christos Dimoulas. 2023. How Profilers Can Help Navigate Type Migration. *PACMPL* 7, OOPSLA (2023), 544–573.

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

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

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

[35] 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

[36] 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.

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

[38] 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

[39] Mohammad Wahiduzzaman Khan and Sheng Chen. 2024. Gradual Typing Performance, Micro Configurations and Macro Perspectives. In *TASE*. 261–278.

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

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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

[47] 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

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

[49] 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

[50] 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

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

[52] 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

[53] 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.

[54] 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

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

[56] 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

[57] 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

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

[59] 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.

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

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

[62] 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

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

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

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

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

[67] 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

[68] 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

[69] Herbert A. Simon. 1947. *Administrative Behavior.* MacMillan.

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

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

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

[73] 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.

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

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

[76] 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.

[77] 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

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

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

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

[81] 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.

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

[83] 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.

[84] 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

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 [34, 74]. The following benchmarks required changes: acquire, kcfa, snake, suffixtree, synth, take5, tetris, and zombie.

In short, 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 library/adaptor boundary instead of the three library/client boundaries. This kind of attribution is bad for the rational programmer because it cannot modify the adaptor. Only the library and client modules can get types.

The change 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 and provides them safely; and

(4) modify the clients to import from the newly-created submodules rather than the top level.

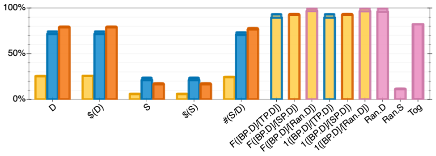
The submodules attach client-specific names to contracts. They do not change run-time behavior.

B RESULTS FROM INDIVIDUAL BENCHMARKS

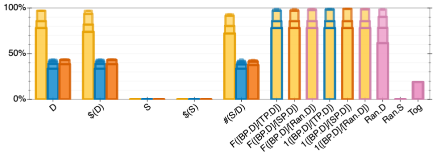
This appendix looks into how closely the trends from the general results apply to individual benchmarks. Specifically, it breaks down Figures 2 to 4 by benchmark.

Figure 2, which summarizes the successes and failures across all benchmarks, suggests that *second-chance* strategies that combine the *boundary* profiler with *random* perform best. The results for 8 of the 16 benchmarks match this trend:

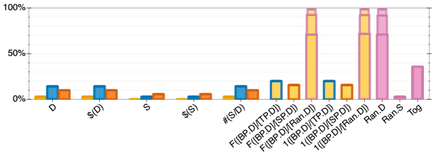
lnm



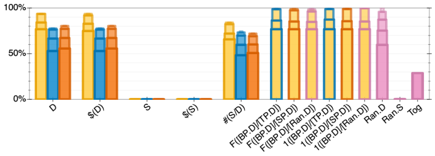
suffixtree



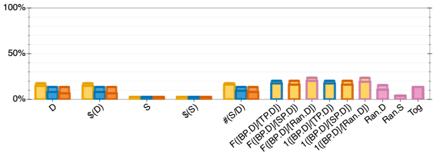
morsecode



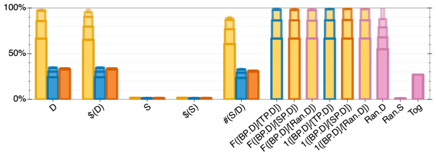
kcfa



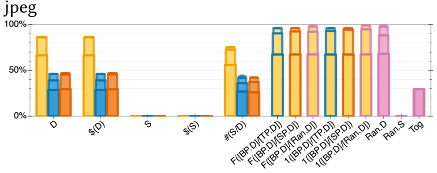
zombie



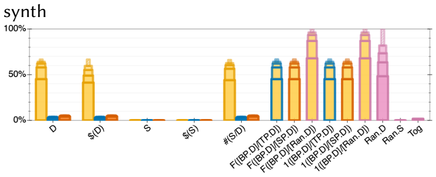
tetris



jpeg



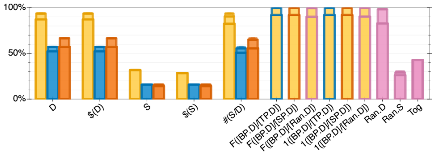
synth



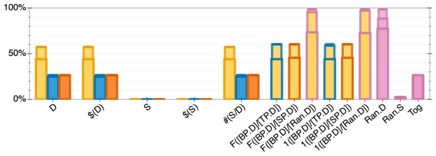
In all of these benchmarks, the strategies similar to *boundary deep* (including *boundary deep* itself, *boundary cost-aware deep*, *configuration-aware*, and the *second-chance* strategies) do better than all others.

However, two of the benchmarks differ from this trend:

fsm

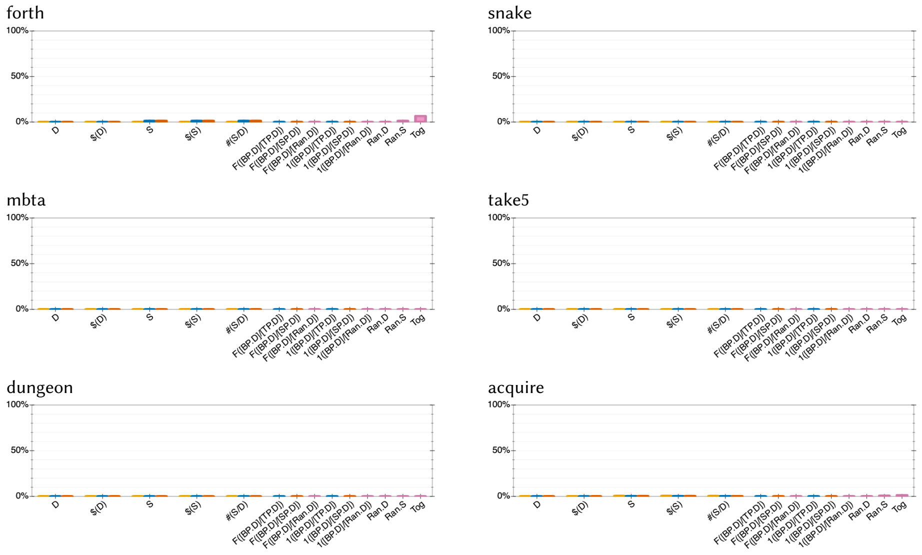


fsmoo



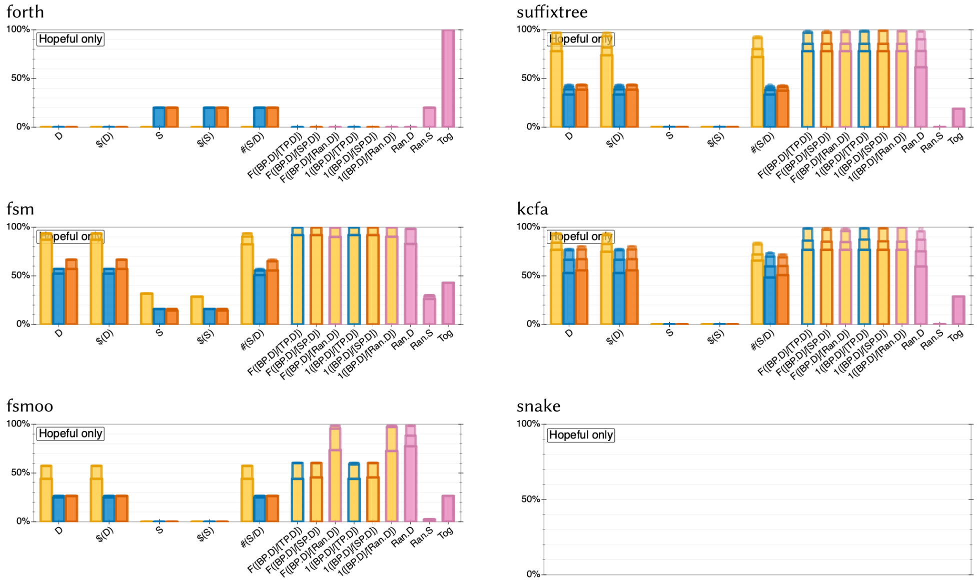
Neither fsm nor fsmoo has a *second-chance random* strategy as their best. For fsm, strategies that resort to the *statistical* profiler when the *boundary* profiler fails do better than those that resort to *random*. For fsmoo, *random deep* on its own outperforms every other strategy when it comes to strict success, although the *second-chance random* strategies still do better on 1-loose.

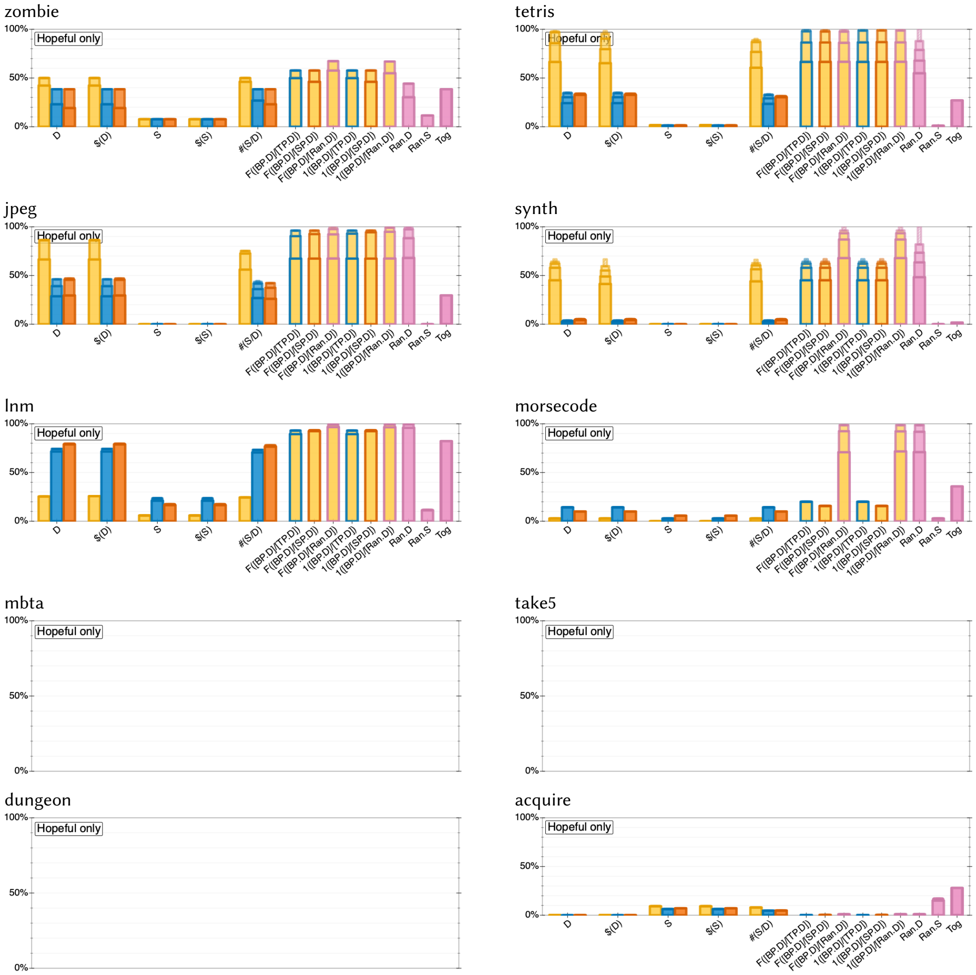
The remaining six benchmarks exhibit pathological obstacles:



The lattices for snake, mbta, take5, and dungeon have zero hopeful performance-debugging scenarios, while acquire's lattice contains around 4-5 % hopeful scenarios (see Table 9). Because a developer does not know the complete migration lattice and therefore cannot predict whether a scenario is hopeful, it is important to remember that these pathological cases exist, and every navigation that a developer embarks on may be one that has no chance of reaching performance parity with the untyped configuration.

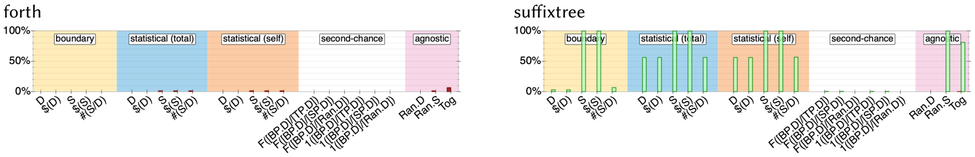
When we restrict the starting points to hopeful scenarios only, the individual benchmarks' results look like this:

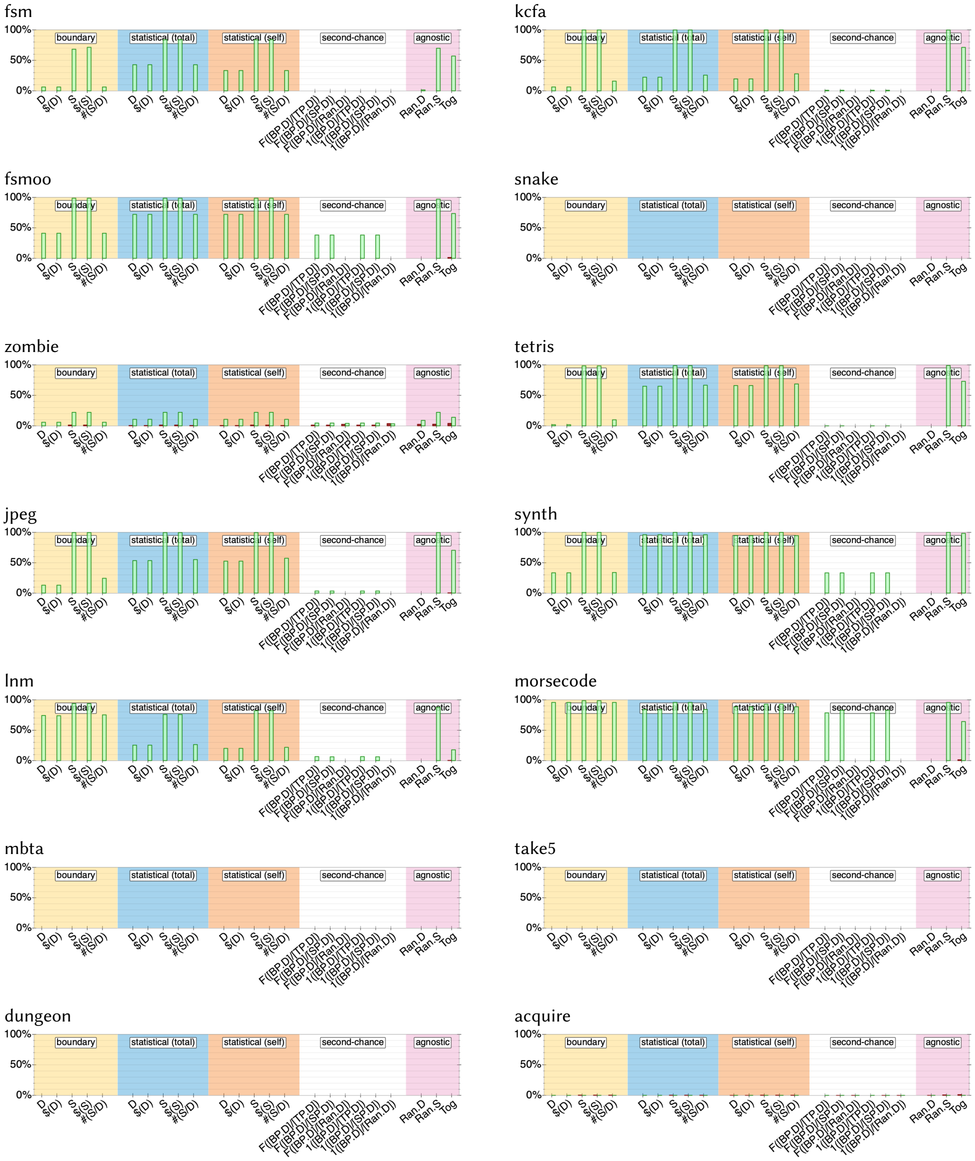




Four of these are completely hopeless: snake, mbta, take5, and dungeon. Hence, their plots are empty, and they are not considered as part of the averages in the Figure 3. Another three (forth, zombie, and acquire) have have a large number of hopeless scenarios (about 94 %, 65 %, and 96 %, respectively). For all notions of success, these three benchmarks' success rates are greater for hopeful-only scenarios than for all scenarios, but exactly the same trends appear. In other words, the success rates are scaled upward but retain their shape.

We now revisit the head-to-head comparisons in Figure 4, breaking down how the *random one-try* strategy fares against the rest by individual benchmark:



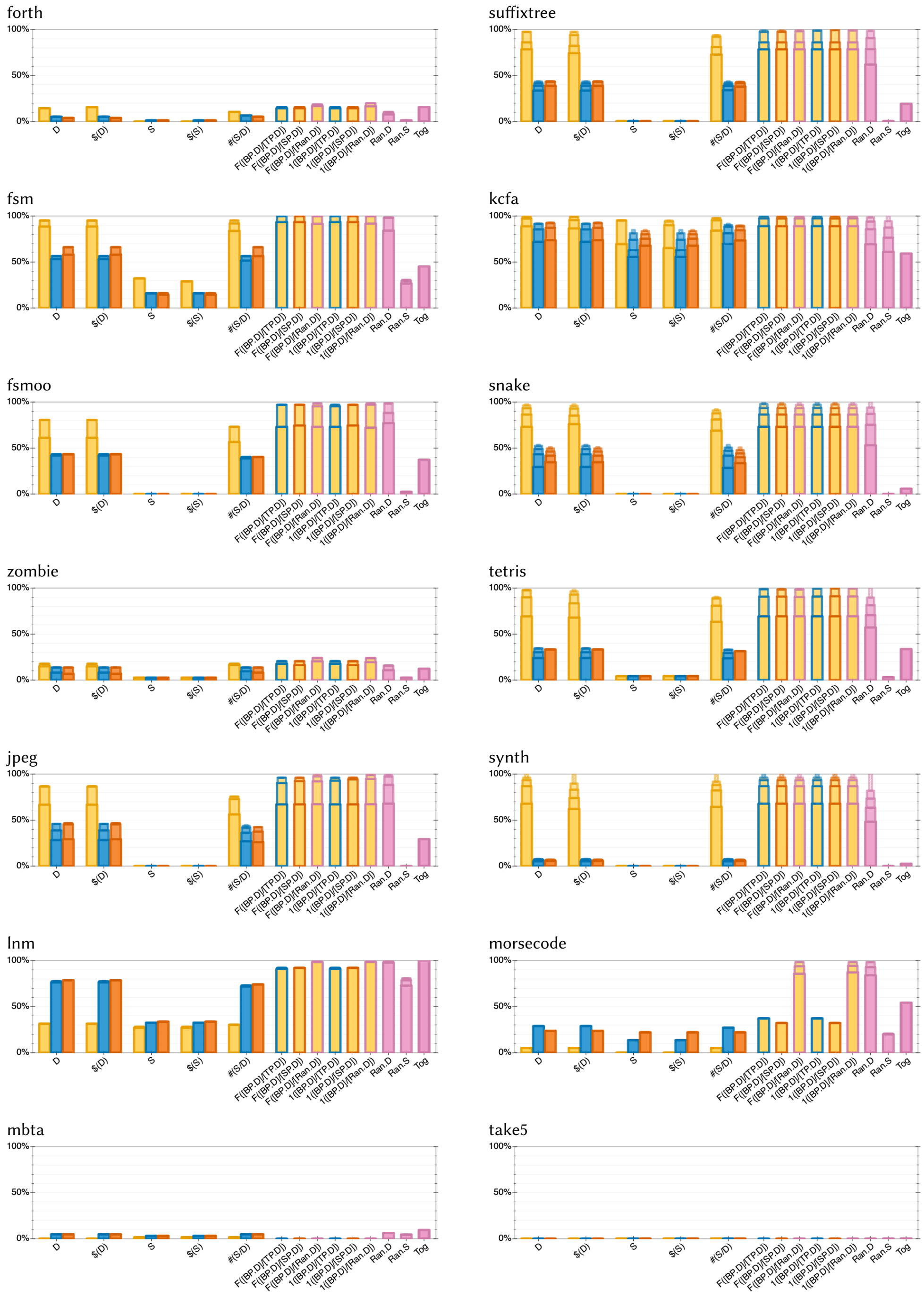


In alignment with Figure 4, *random one-try* outperforms all other strategies in most benchmarks. The two exceptions are forth and zombie.

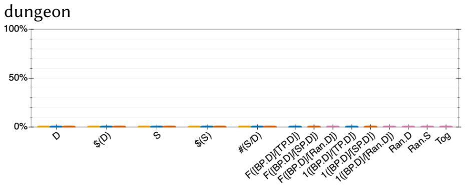
C RESULTS FROM INDIVIDUAL BENCHMARKS FOR ALTERNATE T-VALUES

This appendix visualizes how relaxing the maximum acceptable performance degradation T affects the success rates of the strategies for individual benchmarks. Specifically, it breaks down Figures 6 to 9 by benchmark.

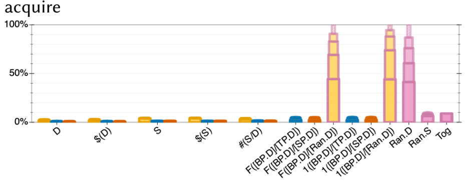
Here are the results for each benchmark when T is relaxed to 1.1:


forth
suffixtree
fsm
kcfa
fsmoo
snake
zombie
tetris
jpeg
synth
lnm
morsecode
mbta
take5
100%
50%
0%
D
$(D)
S
$(S)
#(S/D)
F([BP.D]/[TP.D])
F([BP.D]/[SP.D])
F([BP.D]/[Ran.D])
1([BP.D]/[TP.D])
1([BP.D]/[SP.D])
1([BP.D]/[Ran.D])
Ran.D
Ran.S
Tog

dungeon

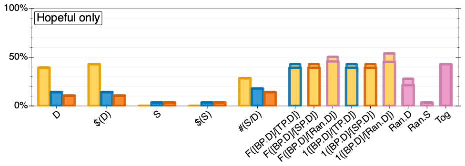


acquire

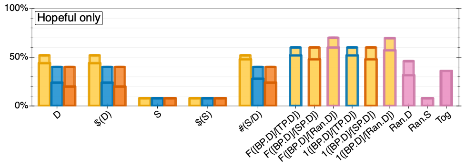


When $T = 1.1$, five benchmarks contain hopeless scenarios: dungeon, forth, mbta, zombie, and take5. Of these, only dungeon is completely hopeless, and the other four are like so:

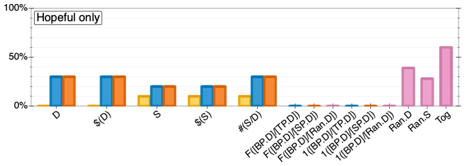
forth



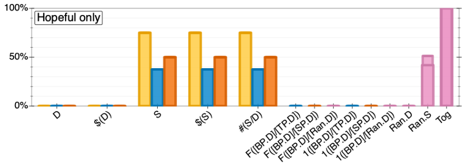
zombie



mbta

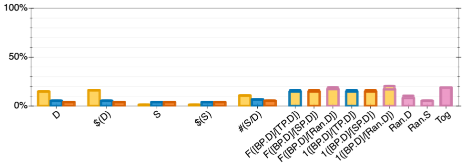


take5

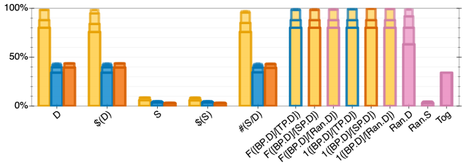


Here are the results for each benchmark when T is further relaxed to 1.5:

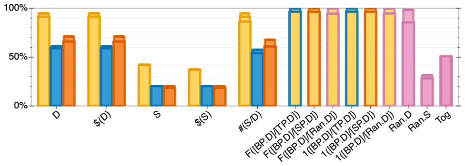
forth



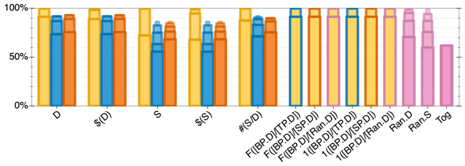
suffixtree



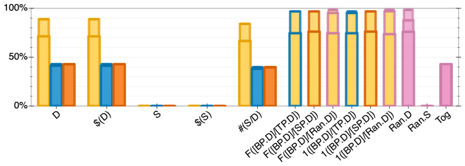
fsm



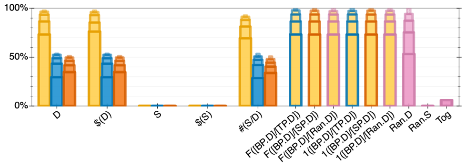
kcfa



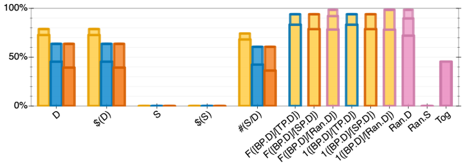
fsmoo



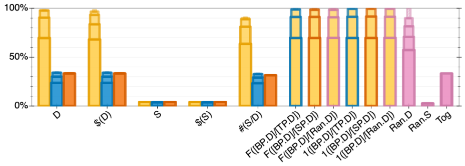
snake



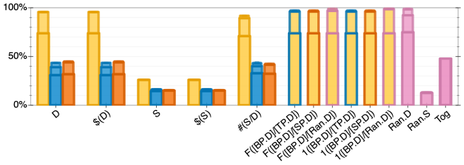
zombie



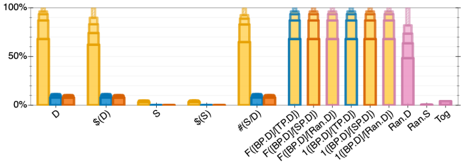
tetris



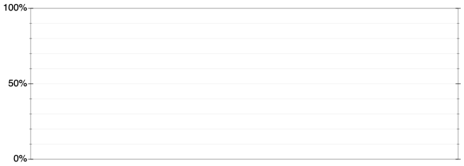
jpeg



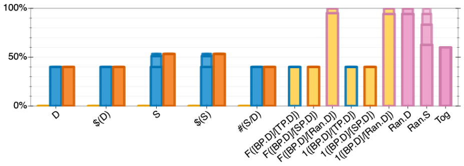
synth



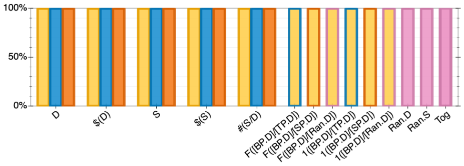
lnm



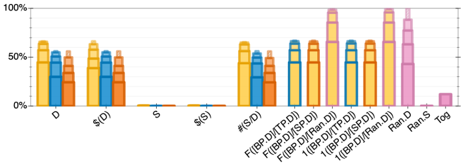
morsecode



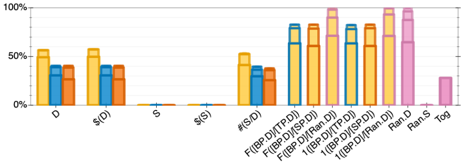
mbta



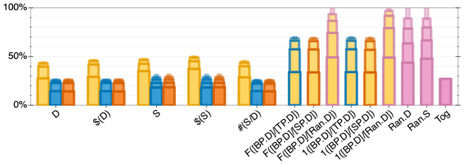
take5



dungeon



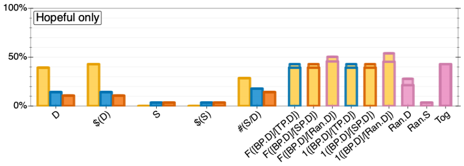
acquire



Since every configuration from lnm has an overhead below 1.5x, lnm no longer offers any performance-debugging scenarios. Its chart is left empty here, and the averages for Figures 8 and 9 are calculated from the other 15 benchmarks only. The mbta also looks quite strange. This is because when $T = 1.5$, mbta only offers 1 performance-debugging scenario, which all of the strategies successfully navigate.

With T at 1.5, only forth contains hopeless scenarios:

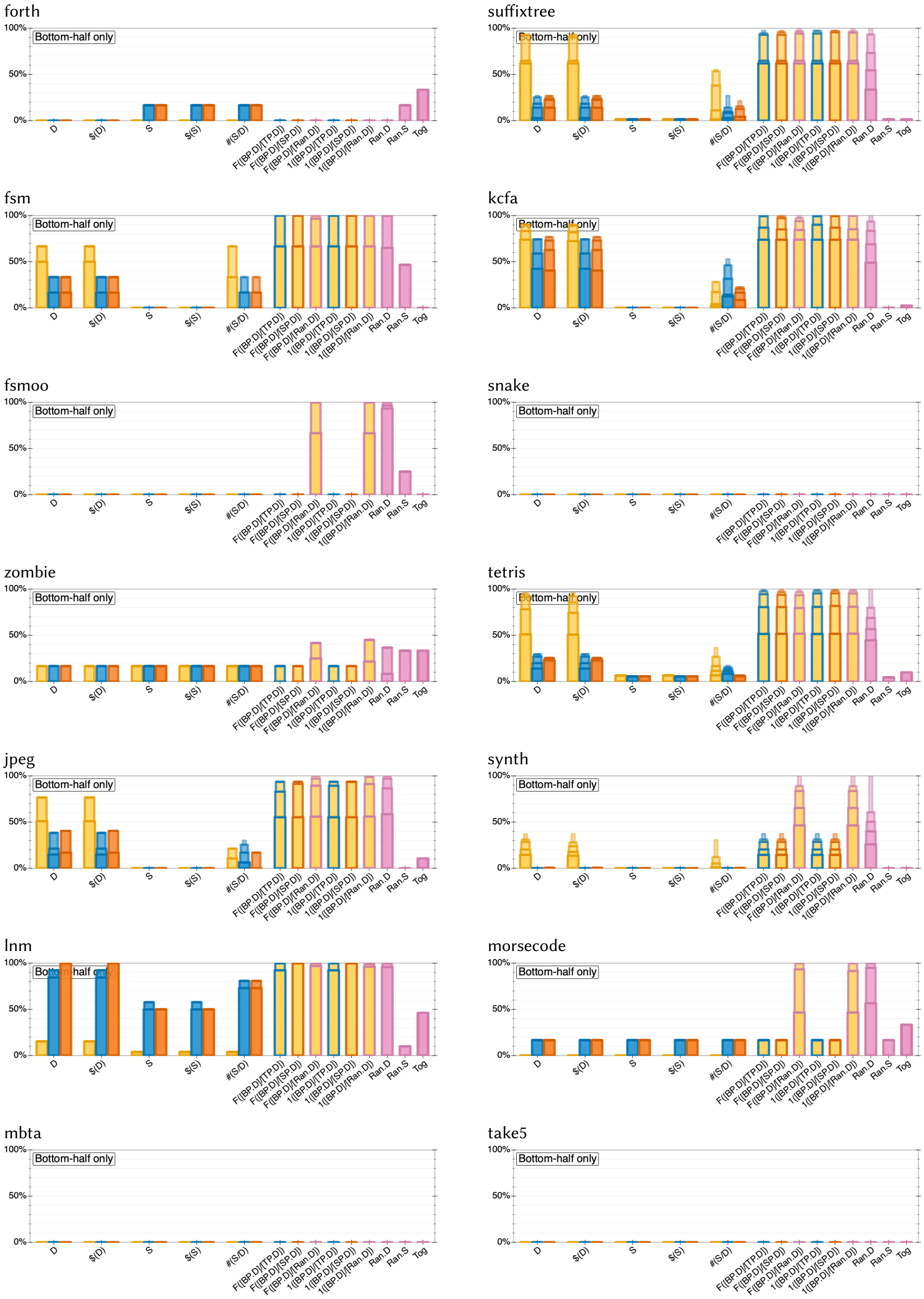
forth

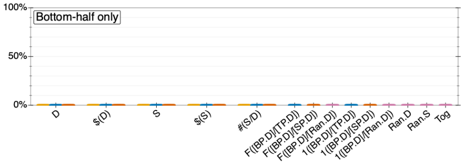


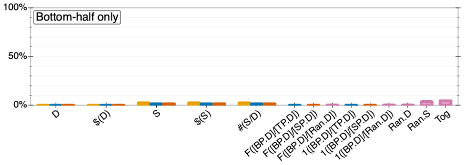
D RESULTS FROM INDIVIDUAL BENCHMARKS, BOTTOM-HALF ONLY

This appendix presents the $T = 1$ success rates of all strategies when drawing scenarios from only the bottom half of each benchmark's lattice. Specifically, it breaks down Figure 10 by benchmark.

Here are the results for each benchmark when scenarios are drawn only from the bottom half of the lattice:

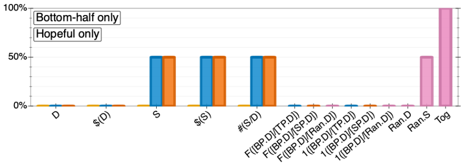

forth
suffixtree
fsm
kcfa
fsmoo
snake
zombie
tetris
jpeg
synth
lnm
morsecode
mbta
take5
Bottom-half only
100%
50%
0%
D
$(D)
S
$(S)
#(S/D)
F([BP.D]/[TP.D])
F([BP.D]/[SP.D])
F([BP.D]/[Ran.D])
1([BP.D]/[TP.D])
1([BP.D]/[SP.D])
1([BP.D]/[Ran.D])
Ran.D
Ran.S
Tog

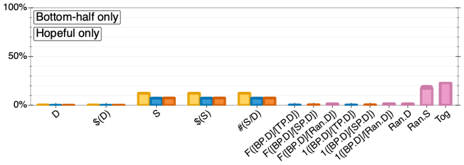

dungeon


acquire

The benchmark displaying the most striking difference from the whole-lattice results is fsmoo, where *random*-based strategies seem to be the only kind that reach any success. The other major difference in trends is that, for most benchmarks, *configuration-aware*'s bottom-half success rate is worse than that of *deep* and *cost-aware deep*, suggesting that its success only ever comes from being identical to *deep* in the upper half of the lattice.

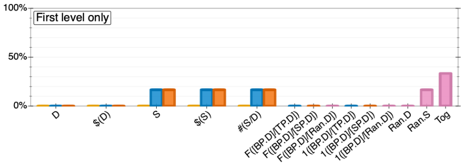
Six benchmarks have hopeless scenarios in the bottom half: snake, mbta, take5, dungeon, forth, and acquire. The first four of these are completely hopeless, but the bottom halves of the lattices for forth and acquire are not:

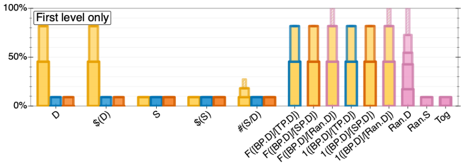

forth

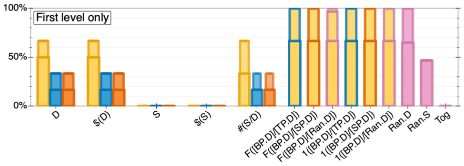

acquire

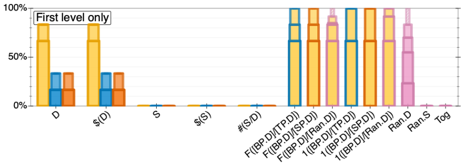
E RESULTS FROM INDIVIDUAL BENCHMARKS, FIRST LEVEL ONLY

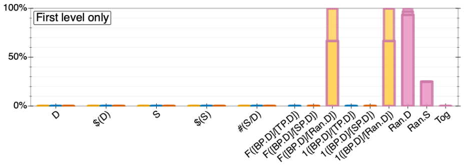
This appendix presents the $T = 1$ success rates of all strategies when drawing scenarios from only the first level of the migration lattice (when exactly one module is typed). Specifically, it breaks down Figure 11 by benchmark.

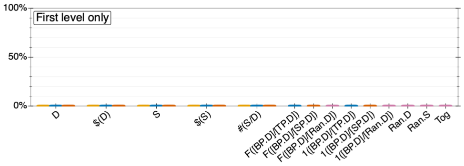

forth

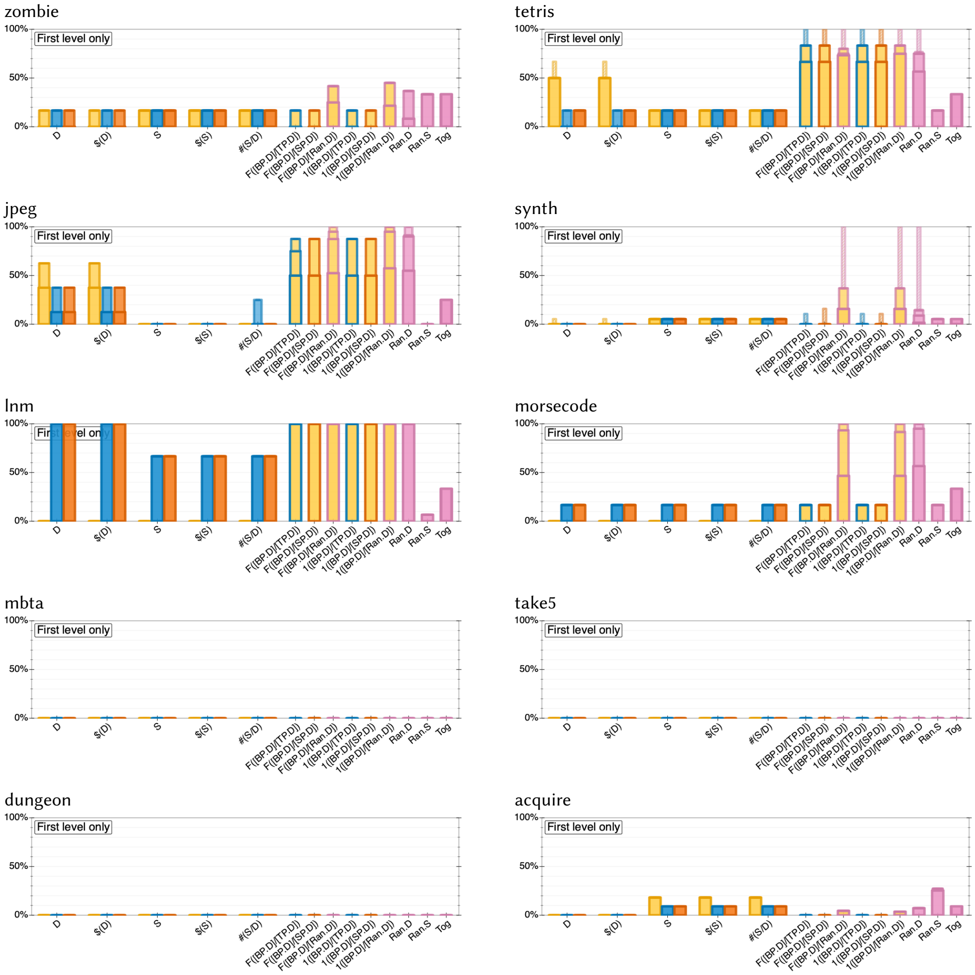

suffixtree


fsm


kcfa

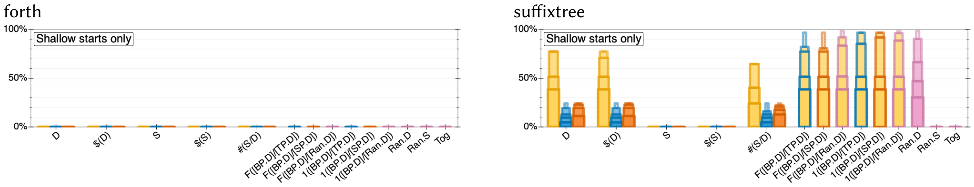

fsmoo

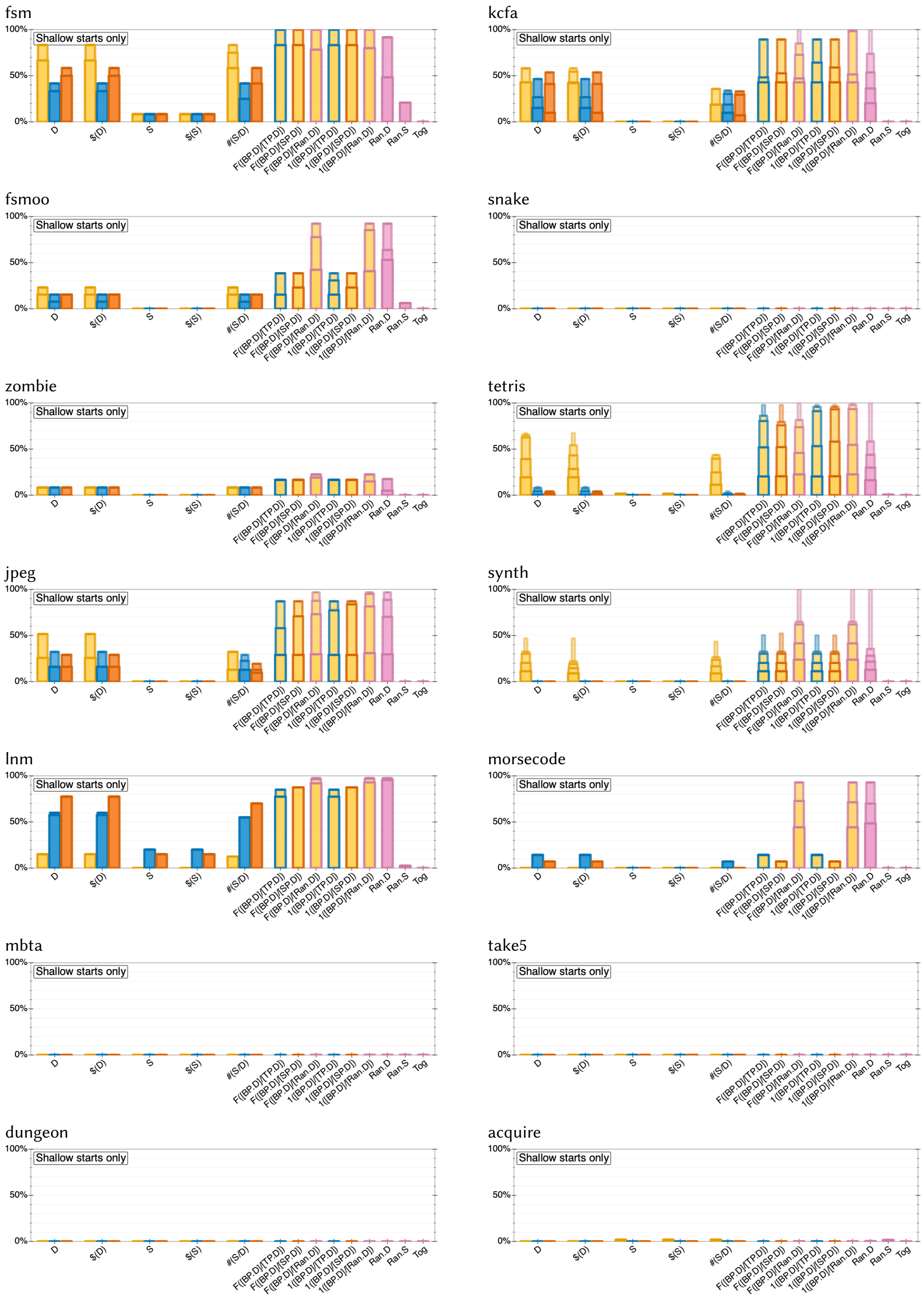

snake



F RESULTS FROM INDIVIDUAL BENCHMARKS, SHALLOW STARTS ONLY

This appendix presents the $T = 1$ success rates of all strategies when excluding starting points that have deep-typed modules. Specifically, it breaks down Figure 12 by benchmark.

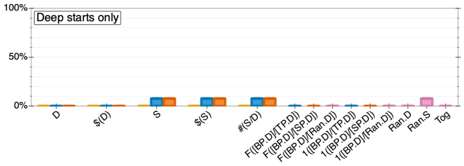



fsm
kcfa
fsmoo
snake
zombie
tetris
jpeg
synth
lnm
morsecode
mbta
take5
dungeon
acquire
Shallow starts only
100%
50%
0%
D
$(D)
S
$(S)
#(S/D)
F((BP.D)(TP.D))
F((BP.D)(SP.D))
F((BP.D)(Ran.D))
1((BP.D)(TP.D))
1((BP.D)(SP.D))
1((BP.D)(Ran.D))
Ran.D
Ran.S
Tog

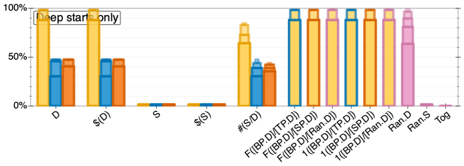
G RESULTS FROM INDIVIDUAL BENCHMARKS, DEEP STARTS ONLY

This appendix presents the $T = 1$ success rates of all strategies when excluding starting points that have shallow-typed modules. Specifically, it breaks down Figure 13 by benchmark.

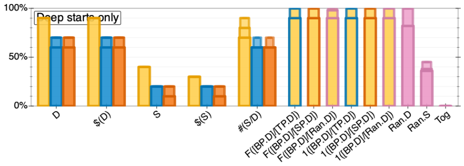
forth



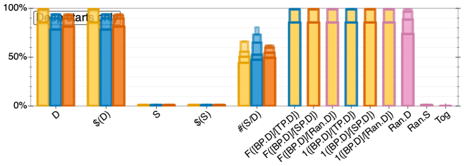
suffixtree



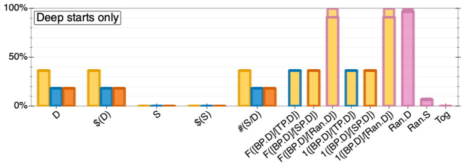
fsm



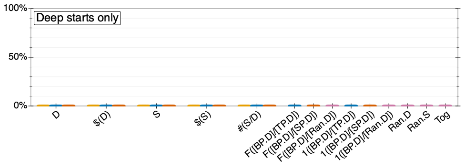
kcfa



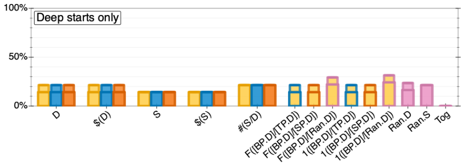
fsmoo



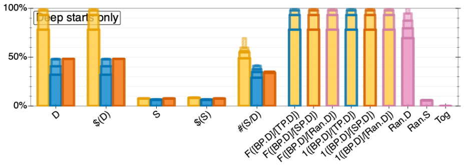
snake



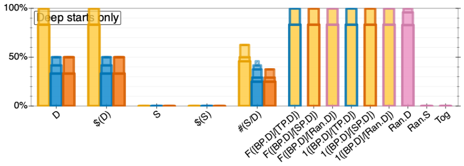
zombie



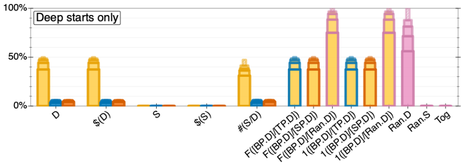
tetris



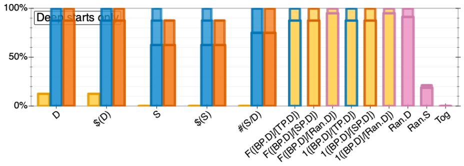
jpeg



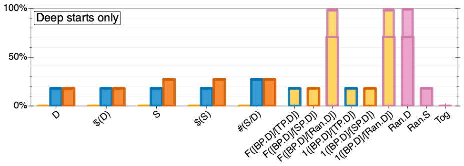
synth



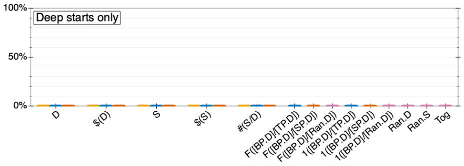
lnm



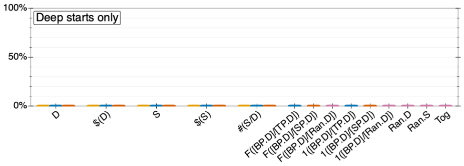
morsecode



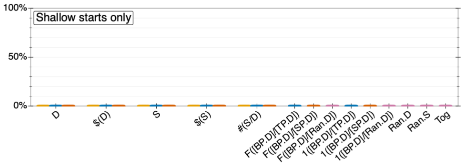
mbta



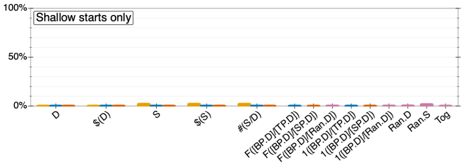
take5



dungeon



acquire


Shallow starts only
D
$(D)
S
$(S)
#(S/D)
F([BP.D]/[TP.D])
F([BP.D]/[SP.D])
F([BP.D]/[Ran.D])
1([BP.D]/[TP.D])
1([BP.D]/[SP.D])
1([BP.D]/[Ran.D])
Ran.D
Ran.S
Tog