Gradual Typing Performance, Micro Configurations and Macro Perspectives

Mohammad Wahiduzzaman Khan and Sheng Chen

UL Lafayette, Lafayette LA 70503, USA
{mohammad-wahiduzzaman.khan1,chen}@louisiana.edu

Abstract. Static typing and dynamic typing have respective strengths and weaknesses, and a language often commits to one typing discipline and inherits the qualities, good or bad. Gradual typing has been developed to reconcile these typing disciplines, allowing a single program to mix both static and dynamic typing. It protects soundness of typed regions with runtime checks when values flown into them do not have required static types. One issue with gradual typing is that such checks can incur significant performance overhead. Previous work on performance has focused on coarse-grained gradual typing where each module (file) has to be fully typed or untyped. In contrast, the performance of fine-grained gradual typing where each single parameter can be partially-typed (such as specifying the parameter as a list without giving element type) has not been investigated.
Motivated by this situation, this paper systematically investigates performance of fine-grained gradual typing by studying the performance of more than 1 million programs. These programs are drawn from seven commonly-used benchmarks with different types for parameters: some parameters are untyped, some are statically typed, and others are partially statically typed. The paper observes many interesting phenomena that were previously unknown to the research community. They provide insights into future research directions of understanding, predicting, and optimizing gradual typing performance as well as migrating gradual programs towards more static.

1 Introduction

In static typing, compilers use types to catch programming errors, provide documentation, and optimize program performance. Static type systems are considered as one of the most successful formal methods. Static typing also has several shortcomings. For example, it requires the program to be complete and free of type errors before it can be run, even though the region that causes the type error may not be covered in a certain execution. It also prevents some commonly used programming idioms for fast prototyping, such as heterogeneous data structures and reflection.

Dynamic typing provides the flexibility that not supported by static typing but offer little static error detection. Overall, these two typing disciplines are complementary, offering different strengthens and having different weaknesses. In early years, a language chose one typing discipline and committed to it. It is impossible to have the advantage of a typing discipline that the language did not choose to follow. For example, it is impossible to have the flexibility of dynamic typing in C, which uses static typing and to have static error detection in Python, which uses dynamic typing.

In real-world development, it is often desirable for a single language to support both static and dynamic typing. There have been two different typing disciplines developed for this purpose, *optional typing* and *gradual typing*. In both disciplines, type annotations can optionally be added to variables, parameters, and return values to specify their types, allowing for static type checking. However, unannotated code retains the flexibility of dynamic typing, enabling rapid prototyping and exploration without the burden of explicit type declarations. Both have attracted enormous attention, from both industry and academia.

The main difference between optional typing and gradual typing is that the latter performs runtime type checks while the former does not. There are many advantages of performing runtime type checks, including ensuring the soundness of typed code, early detection of runtime type errors, and precise blaming of code that violates runtime errors. The main issue with runtime type checks is, however, such checks can significantly slowdown program performance.

The performance problem has been well known. Several studies have conducted in-depth investigations of gradual typing performance [29,13]. These studies have observed that sound gradual typing incurs significant performance overhead, in some case the slowdown could be more than 100 times. However, such studies have focused on gradual typing where type additions are supported only at the granularity level of modules. Specifically, if developers wanted to add type annotations to their codebase, then they have to add type annotations to all functions in a module or they have to leave the module completely untyped. For a codebase with n modules, 2^n different *configurations* may be created, where each configuration adds type annotations to a certain subset of all n modules. Such a gradual typing system has been added to Racket to create Typed Racket.

However, no studies have investigated the performance characteristics of *micro* gradual type systems, where type annotations may be added to any individual function of a module or even a single parameter. Moreover, in practice, the type for each parameter may be a mix of static and dynamic types, rather than fully dynamic (denoting that the parameter is untyped) or fully static (denoting that the full type information for the parameter can be completely decided at compile time). To illustrate, consider the function `reduce` in Figure 1. This function takes three parameters and reduces the list (`lst`) into a single value using the function `f` with the initial value `init`. In this example, the parameter `f` has a dynamic type, written as `Dyn`. This type indicates that the type of `f` can not be known at compile time, and its usage should not be type checked statically. The type for `init` is `Int`, which is fully static. The type for `lst`, `List(Dyn)`, is a partially static type. This type specifies that the parameter `lst` will be used as a list without any restrictions on the element type of the list.

Without a clear understanding of the performance landscape of micro gradual typing, a few questions remain unanswered. First, how to evaluate the performance of micro gradual typing and specific benchmarks? The full configuration space of using micro gradual typing is very huge. For a program with n parameters, we are able to generate 2^n configurations, where the type for each parameter is a `Dyn` or a fully static type. We refer to such configurations as *outlying configurations*. In addition, we are able to generate more configurations where the type for some parameter is a mix of

```
def reduce(f:Dyn, lst:List(Dyn), init:Int):
  result = init
  for i in range(len(lst)):
    result = f(result,lst[i])
  return result
```

Fig. 1: A gradual program in Python type hint syntax that mixes different kinds of type annotations.

`Dyn`s and static types. We refer to such configurations as *intermediate configurations*. For example, `reduce` in Figure 1 is an intermediate configuration because the type for `lst` contains both static type information (`List`) and a `Dyn`. However, had we changed the type of `lst` to `Dyn` or `List(List(Int))`, then `reduce` is an outlying configuration. The number of intermediate configurations quickly doubles the number of outlying configurations, depending on the type structure of each parameter. For example, if the static type for a parameter is `Tuple(Int,Float,List(Int))`, then in an outlying configuration, the parameter may be assigned eleven different types, with two possibilities for the first component of the tuple, two for the second component, and three (`Dyn`, `List(Dyn)`, `List(Int)`) for the third component[1].

Given this sheer number of possible configurations, how to measure the performance of gradual programs? It is obviously infeasible to measure all configurations. Sampling is therefore necessary. The question is, what sampling strategy should be used? Should the samples include both outlying configurations and intermediate configurations or outlying configurations are sufficiently representative? Early work on micro gradual typing performance [32,34,33,4,5] considered only outlying configurations. Should future work on micro gradual typing also sample intermediate configurations for evaluation?

The second question remains unanswered is, what are desirable type annotations for parameters? This question is closely intertwined with the problem of gradual type migration, which studies the challenges and solutions of adding static type annotations to gradual programs. While manually adding type annotations is one possible way for small programs, it does not scale to large programs. Several approaches based on type inference [6,14,3,24,7,27,2,16,22], dynamic approaches [18,9], and machine learning based approaches [17,21,23,1], have been developed. Such approaches often fail to infer most static types for parameters. It is very likely that the user will start from the types added by such type migration tools and make the types more static. The questions are then, how diverse are the run times of the configurations when a single parameter is assigned different types, do the performance keep on increasing when the type for a parameter becomes more static, does the most static type lead to the best performance?

This paper aims to answer these questions through a systematic evaluation of around 1.25 million configurations drawn from six commonly-used benchmarks for gradual typing research. This paper makes the following contributions:

[1] $2 \times 2 \times 3 = 12$, but we need to minus one combination whose all components are static.

```
def reduce1(f:Function([Int,Dyn],Int),
              lst:[Dyn], init:Int):
    result = init
    for i in range(len(lst)):
        result = f(result,lst[i])
    return result

def wider(cw:Int, ci:Dyn) -> Int:
    return max(cw, len(ci))

cont = [[1],[2,3],[4,5,6]]
reduce1(wider,cont,0)
```

(a) `reduce1`, with `lst` type being `[Dyn]`

```
def reduce1(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = f(result, lst[i])
    return result : Int => Dyn

def wider(cw, ci):
    return max(cw, len(ci)) : Dyn => Int

cont = [[1],[2,3],[4,5,6]]
reduce1(wider, cont : Dyn => [Dyn], 0)
```

(b) Cast inserted version of `reduce1`

```
def reduce2(f:Function([Int,Dyn],Int),
             lst:[[Dyn]], init:Int):
    result = init
    for i in range(len(lst)):
        result = f(result,lst[i])
    return result

reduce2(wider,cont,0)
```

(c) `reduce2`, with `lst` type being `[[Dyn]]`

```
def reduce2(f, lst, init):
    result = init
    for i in range(len(lst)):
        result = f(result, lst[i] : [Dyn] => Dyn)
    return result : Int => Dyn

reduce2(wider, cont : Dyn => [[Dyn]], 0)
```

(d) Cast inserted version of `reduce2`

Fig. 2: Two different versions of `reduce` (left) that differ by only one parameter type and their corresponding cast-inserted programs (right). The function type with two parameters whose types are `t1` and `t2` and with return type `t3` is written as `Function([t1,t2],t3)`.

1. It creates a benchmark with around 1.25 million configurations. Such configurations have fine-grained types. With precise type information and corresponding execution times, it facilitates future research in gradual typing.
2. It studies three research questions concerning the representativeness of outlying configurations and performance change as parameter types undergo minor changes.
3. Based on the evaluation results, it makes affirmative answers to the studied research questions. Specifically, the result reveals that outlying configurations are not always representative, the performance can change radically even with small changes in type annotations, and counter-intuitively, the performance often decreases as parameter types become more precise. These answers suggest a better performance evaluation method for future work on gradual typing performance. They also indicate where further research attention is needed to make gradual typing practical.

2 Background

The purpose of gradual typing is to strike a balance between the safety and performance guarantees of static typing and the flexibility and expressiveness of dynamic typing. As such, in gradual typing, a parameter or variable may be assigned a static type if the type expectation of the parameter or variable can be statically determined and the uses of it need to be statically checked. In contrast, if the type can not be determined statically, then the type should remain dynamic. The absence of a type annotation or a `Dyn` for a parameter denotes that the parameter has dynamic type. It is also possible that a parameter is annotated with a partially static type, as `[Dyn]` for `lst` in Figure 2(a).

A static type annotation for a parameter is a protocol for both the internal and external of the function. For the internal of the function, the type specifies the *guarantee* of the type of the parameter to the rest of the function definition. For example, the type for `f` indicates that the return type is `Int`. As a result, inside the function, every call will always return a value of type `Int`. For the external of the function, the type specifies the *expectation* of the corresponding argument to the function. For example, when `reduce1` is called, the first argument must be a function type and the first parameter type and the return type of it must be `Int`.

The argument may be statically typed and matches the expectation of the parameter type. In this case, no runtime checks are needed. Otherwise, runtime checks will be inserted to protect the type annotations, a notion known as enforcing *type soundness*. To illustrate, consider the execution of the program in Figure 2(a). When a gradually-typed program is executed, it is often translated to a program in the underlying language with runtime checks inserted. For example, the gradually-typed language Reticulated Python [32,34] is translated to Python. The translated program for Figure 2(a) is given in Figure 2(b).

In translating the call of `reduce1` (the last line of Figure 2(a)), the type expectation of `f` (`Function([Int,Dyn],Int)`) matches the type of the argument since the type of `wider` is also `Function([Int,Dyn],Int)` (see the caption of Figure 2 for an explanation of type syntax for function types). As a result, no runtime checks will be inserted for `wider`. However, for `lst`, the expectation is `[Dyn]` (based on the type annotation for `lst` in the definition of `reduce1`), and the argument has type `Dyn` (many gradual type systems do not assign a static type to a list because lists can be heterogeneous). As a result, a runtime check, often called a *cast*, is inserted to make sure that `cont` has the expected type for calling `reduce1`. The cast is written as `cont: Dyn => [Dyn]`, expressing that the statically known type of `cont` is `Dyn` but it is used in a context that requires it to have the type `[Dyn]`. A general form of a cast is written as `expr : source_type => target_type`.

Casts may have very different runtime overheads. Basic casts involving primitive types (such as `Int`, `Bool`, and `Float`) and `Dyn` are very lightweight. For example, the cast `e: Dyn => Int` induces very little runtime overhead because a single runtime type check (such as `isinstance(e,int)` in Python) suffices to check if the cast will be successful. However, casts involving other types, such as lists and functions, can be very expensive. The reason is that such casts can not be verified at where they appear. To illustrate, consider a cast `f:Dyn => Int -> Bool`. This cast means that `f` should return a `Bool` value whenever it is called with an `Int` value. It is impossible to measure whether

`f` satisfies the cast at its occurrence location because in dynamic languages a function may return values of different types for values of the same type.

Instead, for such a cast, a proxy needs to be created to make sure that it has expected type at each call site. Continuing the example from the previous paragraph, when `f` is called, the proxy for `f` checks whether the argument has the type `Int` and the return type of `f` has the type `Bool`. Creating proxies and checking the argument and return types for each call involves more significant overheads.

Such casts are the main reason that sound gradual typing radically slows down program performance. Earlier work [29] observed this phenomenon when types are added to a single module (file). This work investigates the impact of changing the type annotation of a single parameter on program performance. To illustrate how performance may be affected due to a type change, consider the program in Figure 2(c). The only difference (with a gray background) compared to Figure 2(a) is that the type annotation for `lst` is changed to `[[Dyn]]`. The single type change, however, led to the changes of two inserted casts, shown in Figure 2(d) with a gray background. One of these casts is inside a `for` loop, and it in fact induces high overhead for this program. We measured the performance of these programs and observed that the performance for program in Figure 2(c) doubles that in Figure 2(a).

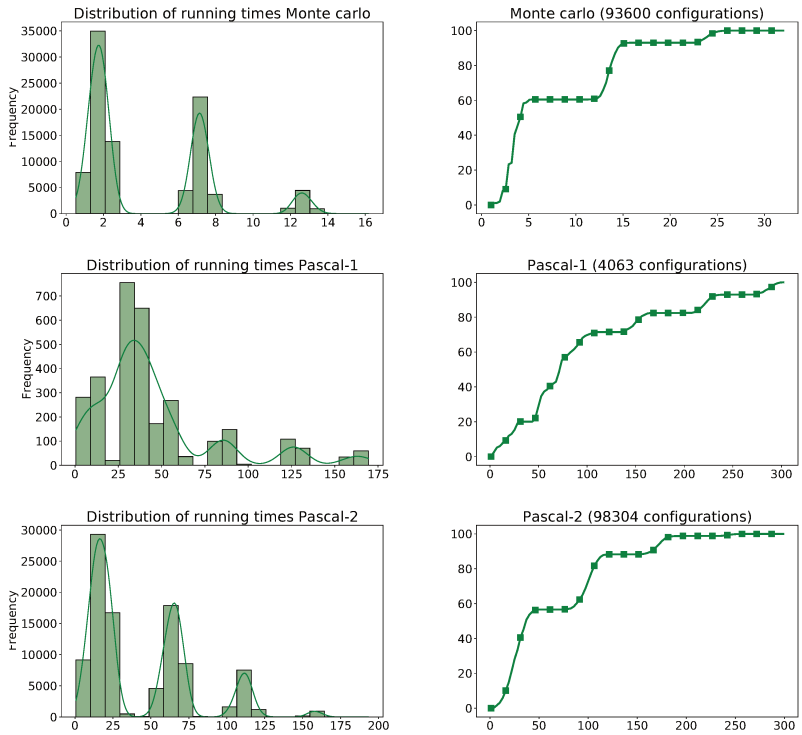

Fig. 3: The execution time distribution for each benchmark.

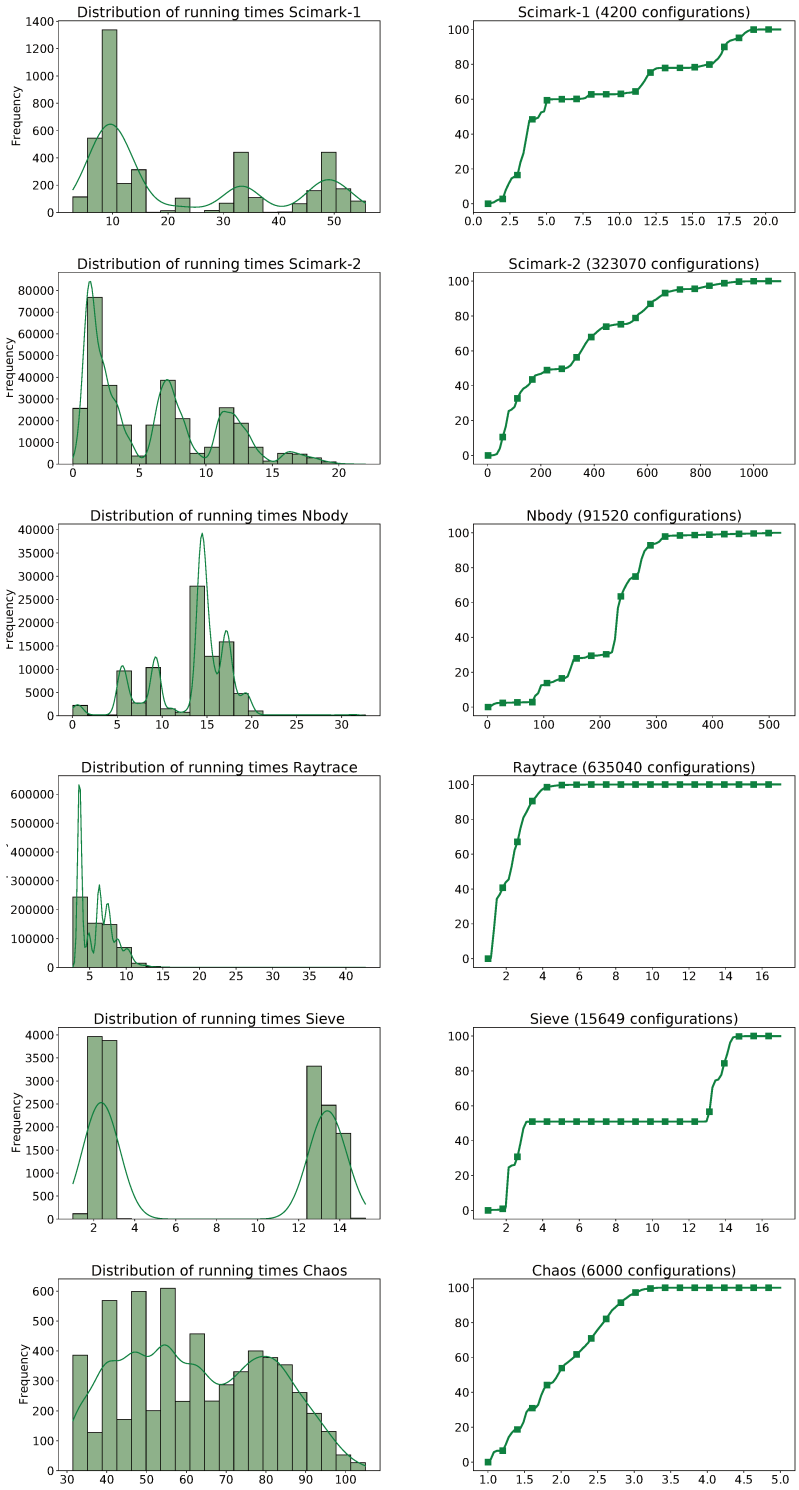

Fig. 4: The execution time distribution for each benchmark (continued)

Benchmark	LOC	# of functions	# of pars	# of typed pars	# of configurations
Monte Carlo	90	4	9	9	93600
Pascal-1	70	7	19	15	4062
Pascal-2	70	7	19	15	98304
Scimark-1	65	5	22	17	4602
Scimark-2	65	5	22	17	323070
Nbody	195	4	21	18	91525
Raytrace	455	21	94	67	635040
Sieve	56	9	22	21	15361
Chaos	271	22	42	29	6000

Table 1: Python benchmarks used for performance evaluation. The last column gives the number of configurations generated for the corresponding benchmark. We generated two datasets for Pascal and Scimark to investigate how the size of dataset affects evaluation results.

3 Benchmarks and Evaluation Protocol

For the purpose of evaluation, we consider seven benchmarks. They are mainly adapted from Python performance benchmark suits and have been frequently in gradual typing research [34,32,4,5]. Table 1 lists some basic metrics of each benchmark, such as number of lines of code, number of functions, etc. Note, the number of typed parameters may be fewer than the number of parameters in a program because not all parameters can receive static types. For a parameter that can be statically typed, we consider different possible types for the parameter, from `Dyn` to most static. To generate a manageable number of configurations for each benchmark, we set a single type for some parameter if the parameter does not have much interaction with the rest of the program. For example, for Nbody benchmark, it has one function defined as follows.

```
def bench_nbody(loops, reference, iterations):
        for _ in xrange(loops):
                ...
```

Here `loops` can be given two possible types: `Dyn` or `Int`. Since `loops` does not interact with the rest of the code except for being used in `xrange`, we consider only assigning `Int` to `loops`. To test the validity this idea, we generate many pairs of configurations such that in each pair `loops` receive different types. We observed that the execution times are almost exactly the same within each pair. We briefly describe each benchmark below.

Monte Carlo This benchmark is to predict possible outcomes of uncertain events by generating random numbers. It contains complex parameter and return types for several functions. One such type is `Tuple([Float],[Float],[Int]))`.

Pascal The purpose of this benchmark is to evaluate the efficiency and performance of algorithms and functions related to generating Pascal's triangle and permutations in Python. One of the parameter in this benchmark is `matrix`, which is essentially a 2-D array. In our generated configurations, we assigned four different types to this parameter: `Dyn`, `[Dyn]`, `[[Dyn]]`, and `[[Float]]`.

Scimark It has six functions which contains a parameter `arr`. The most static type of `arr` is `Tuple(Int,Int,Int)`. This parameter can be typed in 8 different ways.

Sieve This benchmark implements the search of prime numbers using the idea of a "Steam". A main parameter in this benchmark is `st`, which represents the current state of the stream. Its most static type is class `Stream`.

Nbody. This benchmark simulates the movement of celestial bodies under gravity. It has several complex parameter types. For example, one of them is `[Tuple(Tuple([Float], [Float], Float), Tuple([Float], [Float], Float))]` There can be multiple ways with `Dyn` combinations to type this parameter.

Raytrace This benchmark is for simulating lighting for games. Its type structure is similar to that of `Scimark`.

Figures 3 and 4 depict the runtime distribution of generated configurations for each benchmark. For the left column in each figure, the x-axis is the execution time in seconds and the y-axis is the number of configurations. Each image in the right column presents the performance slowdowns compared to the fastest configuration in each benchmark. The x-axis is the slowdown and the y-axis is the percentage of all configurations. A point of (x,y) on the curve means that x% of all configurations have a slowdown of smaller than y compared to the fastest configuration.

From these figures, we can observe that the execution times are very different within and across benchmarks. The distributions are also very different, some have more spread-out times while others are clustered. The number of clusters and the distances between them are also quite different. These factors encourage the validity of the observations we make in this paper.

Evaluation protocol We use Reticulated Python [32,34], a gradual typing implementation for Python, to measure the execution time. The experiments are run on a system equipped with Intel(R) 533 Core(TM) i9-9900K CPU @ 3.60GHz, 8 Core(s), and 32GB RAM. Each measured time is an average of 10 runs.

4 Representativeness of Outlying configurations

While previous work [4,34,33] on gradual typing performance has largely focused on outlying configurations, it is an interesting question to know if such configurations are indeed representative of the whole benchmark.

For this investigation, we first separate all configurations into outlying configurations and intermediate configurations. A configuration is outlying if the type for each parameter in the configuration is a `Dyn` or a fully static type. Otherwise, a configuration is intermediate.

We depict the temporal distribution of runtimes for outlying and intermediate configurations. The kernel density (KD) plots (the left column of the figures) illustrate the divergence in runtime characteristics between outlying and intermediate configurations. An inspection of the KD plots reveals that the multimodal nature of both outlying and intermediate configurations is characterized by the presence of multiple peaks. The box plots (the right column of these figures) provide insights into the central tendency of each distribution. Notably, outlying configurations exhibit a tendency to clusters closer to their mean, while intermediate configurations display numerous outliers distributed far from the mean.

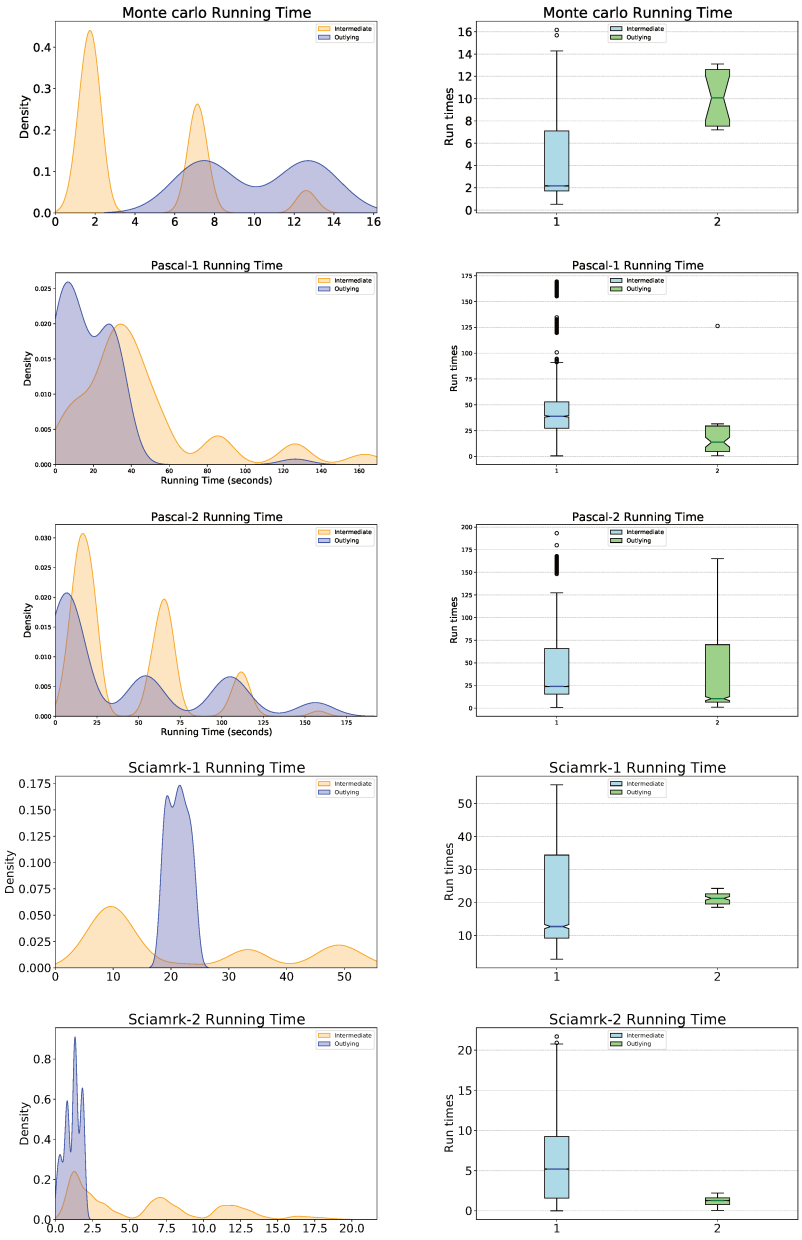

Fig. 5: Execution time distribution of outlying and intermediate configurations

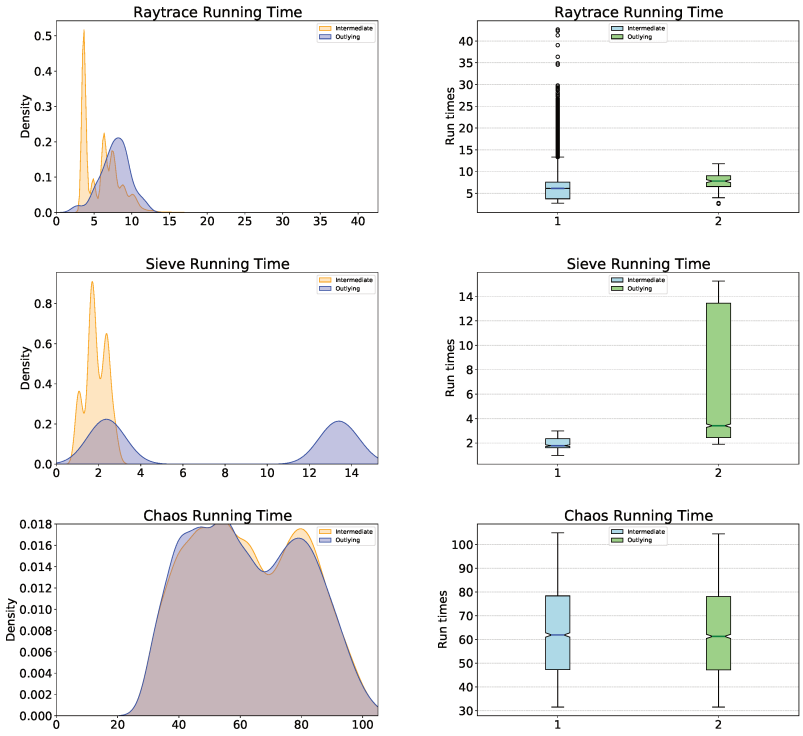

Fig. 6: Execution time distribution of outlying and intermediate configurations (continued)

Based on these figures, we can observe that, except for Pascal-2, Raytrace, and Chaos, the outlying configurations do not represent the whole benchmark well. Moreover, the reason why they are not representative vary across benchmarks. For example, for Monte Carlo, outlying configurations represent configurations with large execution times only while for Pascal-1 they represent only those with small execution times.

Overall, these figures suggest that future work on gradual typing performance should sample from outlying as well as intermediate configurations to make sure the result is representative.

5 Time Variations of Intermediate configurations

This section investigates how diverse are execution times of intermediate configurations that differ by type annotations for a few parameters. Understanding the diversity is an important problem that has both practical and theoretical implications.

In practice, type migrations are often done in small steps or with the help of type migration tools. It is often common that such tools infer only part of the type information, infer too specific type, or contain incorrect types, particularly when the static type

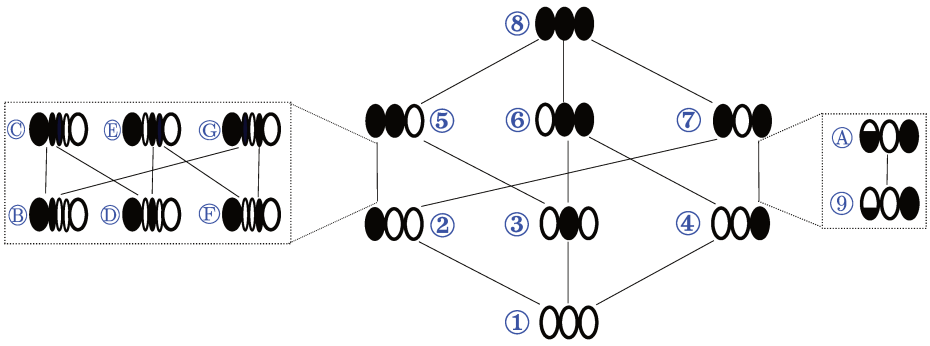

Fig. 7: The lattice, which has 8 outlying configurations (① through ⑧), for a program with three parameters. Each configuration is represented as three ovals, which each oval indicates if the corresponding parameter is typed (filled) or not (unfilled). Each solid line connects a less precise configuration (lower in the lattice) to a more precise one (higher in the lattice). Assume the fully static type of the first parameter is `[[Float]]`, two intermediate configurations, one with the type of first parameter being `[Dyn]` (⑨) and the other being `[[Dyn]]` (Ⓐ), may be added between ④ and ⑦. Likewise, assuming the full static type of the second parameter is `Tuple(Int,Float,Bool)`, then eight intermediate configurations (Ⓑ through Ⓖ) can be generated between ② and ⑤. In Ⓑ through Ⓖ, the three ovals in the middle indicate whether the three components of the second parameter have static types or not.

is complex. In this case, it is likely that the developer will fix the type annotations, make them more specific, or change them to new types. From the performance perspective, there are several related questions. First, how such changes will affect the performance of the program? Will the performance largely stay the same because the change of the type is relatively minor compared to the type annotations for the whole program or will such a small change already lead to significant performance swings?

Second, if the performance indeed changes radically due to such small type changes, is the performance getting better or worse? The answer to this question provides useful guidance for type migration. For example, if slightly making the type of a parameter more specific decreases the performance, should the user make it even more specific or reverse the type addition to restore the performance?

The answers to these questions can help both the users and researchers of gradual typing. For the former, these answers will help develop useful guidance with program migration, making programs more static while aware of the performance landscape. For the latter, these answers reveal the real challenges of harmonizing program migration and performance issues, indicating where the research is needed to make gradual typing practical.

To answer these questions, we need to first find out groups that have different type annotations for a specific number of parameters. To illustrate, consider the lattice in Figure 7. From this lattice, we can extract two groups. The first group contains four configurations (④, ⑦, ⑨, and Ⓐ), differing types in the first parameter. The second

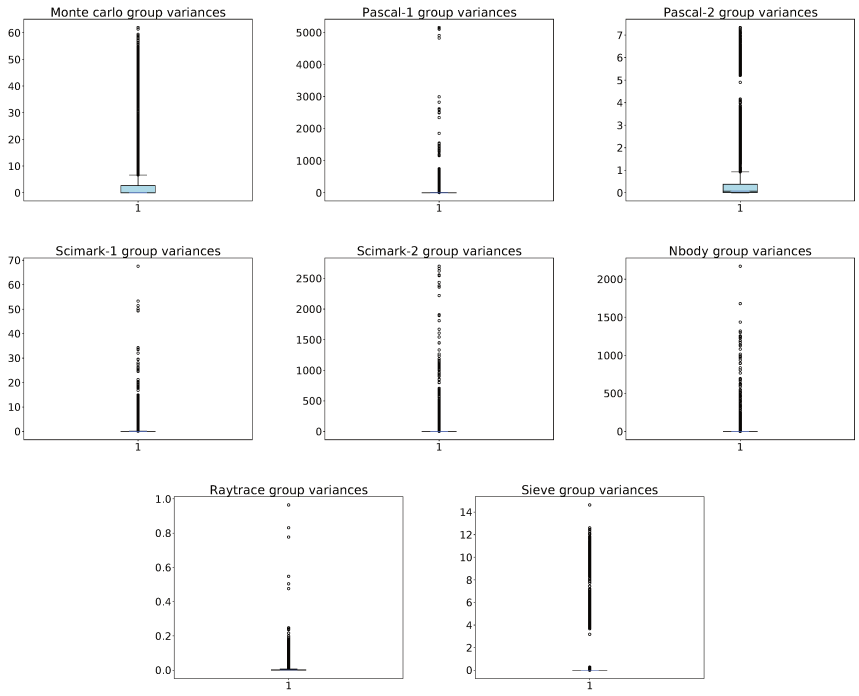

Fig. 8: Benchmark's group variance distribution

group contains 8 configurations (②, ⑤, and Ⓑ through Ⓖ), differing types in the second parameter.

Given a group, we want to measure how diverse are the execution times of the configurations in that group. We use *variance* to measure the variability of execution times. Variance gives us an idea of how much the execution times deviate from the average or mean runtime for that group. We calculate variance for each group in the following steps. First, we compute the mean runtime for each group of programs. This is achieved by summing up all the execution times within a group and dividing the total by the number of configurations in that group. Second, for each individual runtime in the group, the squared difference between it and the mean runtime is determined. Squaring these differences ensures that negative and positive deviations do not cancel each other out, providing a more accurate representation of dispersion. Third, the average of all the squared differences calculated in the previous step is determined. This average value represents the variance of the runtimes within the group.

Based on the calculation, a larger variance indicates greater spread or variability of runtimes from the mean, suggesting a less consistent or stable performance across the programs within the group. Conversely, a smaller variance signifies that the runtimes are closer to the mean, indicating less variability and potentially more consistent performance.

Table 2: Benchmarks groups variance information

Benchmark	# of Groups	Avg group size	min variance	average variance	max variance
Monte Carlo	57120	8	0.0	2.58	61.976
Pascal-1	4096	4	0.0	30.97	5153.21
Pascal-2	110376	7	0	0.27	7.33
Scimark-1	8226	3	0.0	0.83	67.57
Scimark-2	446875	5	0.0	3.56	2700.23
Nbody	43969	8	0.0	2.87	2170.60
Raytrace	10061	6	0.0	0.06	0.96
Sieve	71888	3	0.0	0.82	14.62

Table 3: Benchmarks group ordering information

Benchmark	# of Groups	Avg group size	% of increasing	% of decreasing	% of neutral
Monte Carlo	57120	8	34.69%	21.26%	44.04%
Pascal-1	4096	4	36.54%	16.62%	46.82%
Pascal-2	110376	7	12.15%	16.28%	71.56%
Scimark-1	8226	3	66.44%	21.34%	12.20%
Scimark-2	446875	5	11.29%	41.89%	46.81%
Nbody	43969	8	67.18%	4.55%	28.26%
Raytrace	10061	6	32.85%	14.34%	52.79%
Sieve	71888	3	88.37%	11.45%	0.16%

Table 2 gives the information about the number of groups for each benchmark, average group size, the minimum, average, and maximum variance for all the groups within a benchmark. Figure 8 depicts the variances of all groups within each benchmark. Based on this figure, it becomes evident that certain benchmark groups exhibit notably higher variances compared to others. For instance, despite having similar code structures, Pascal-1 and Pascal-2 demonstrate significantly different variances, with Pascal-1 displaying a notably higher variance. Conversely, in the case of Scimark-1 and Scimark-2, which also share similar code structures, Scimark-1 exhibits considerably higher runtime than Scimark-2.

Based on these results, we can conclude that, with a small change of type annotation for a single parameter, the performance can be very different. This indicates the importance of finding the right type for each parameter to achieve good performance.

Given that each groups has a lot of variance, it is interesting to study if the performance is increasing, decreasing, or a mix of them. For such a study, we first order all configurations within a group based on the precision of the type. For example, as the type for the first parameter in ④ (Figure 7) is `Dyn` while that for ⑨ is `[Dyn]`, we say ⑨ is more precise than ④. After ordering, ⑦ is the most precise, followed by Ⓐ, ⑨, and ④. It is possible that not all configurations in a group can be ordered, such as Ⓑ and

Ⓔ. For such groups, we consider only the path that configurations can be fully ordered along the path.

After ordering each group, we can easily decide if the execution time is decreasing, increasing, or a mix. Table 3 summarizes the result for this study. In the table, the last three columns calculate the percentage of all groups whose execution times are increasing, decreasing, or neutral. Note, if a group has an increasing execution time, the performance degrades as configurations become more precise. Based on the results, we can conclude that the performance often decreases as programs become more precise. Fortunately, there are exist groups whose performance increases as program become more precise. This indicates that future research is needed that takes into consideration both performance and program migration. In particular, it is critical to find parameters such that making them more precise also increases the performance.

6 Related Work

This paper studies the performance of micro gradual typing, where a partial or full static type annotation may be given to a parameter, return value, or variable. Micro gradual typing has been used in Reticulated Python [34,32], Grift [15], and many others [25,26]. There is another kind of gradual typing, adopted by Typed Racket [30], where the decision of whether adding type annotation or not is made at the granularity of a module. Several studies [29,13] have investigated the performance of this kind of gradual typing. A benchmark for this kind of gradual typing research has also been developed [12]. The goal of these papers are thus quite different from ours. Moreover, this paper also investigates representativeness of outlying configurations and how performance changes as the type for a single parameter experiences changes, which are unique to micro gradual typing.

The overhead of gradual typing is due to the checks inserted for protecting typed regions. Such checks are performed at runtime. In contrast, *optional typing* uses type annotations for performing static type checking to catch more programming mistakes before program are run. TypeScript [10], Flow [8], and type hints for Python[2] fall in this approach. To illustrate the difference between optional and gradual typing, consider the expression `reduce1(add,['c','d'],0)` where `add` is defined as follows and `reduce1` is defined in Figure 2(a).

```
def add(a:Int, b:Int) -> Int:
    return a+b
```

In both optional typing and gradual typing, no static type errors are detected in the expression `reduce1(add,['a','b'],0)`. However, the behavior of runtime error reporting is very different. In optional typing, the runtime error is reported inside the definition of `add`, when `a` receives the value 0 and `b` receives the value `'c'`. In gradual typing, the type error is reported at `lst[i]` within `reduce1` because `lst[i]` has the type `String` while the parameter type of `add` is `Int`. An important design principle of gradual typing is that well-typed programs should not be blamed [28,35] for causing runtime type

[2] `https://docs.python.org/3/library/typing.html`

errors. Since `add` is fully typed, its body should never be blamed for causing dynamic type errors.

A recent user study [31] with programmers has revealed that in practice programmers anticipate type systems to behave like gradual typing. When a runtime error happens, they prefer the error to not be reported within a typed function.

Program migration, which aims to add static type annotations to a dynamic program, and performance understanding and optimization have been two important aspects of gradual typing. Many approaches have been proposed for program migration[6,14,3,24,7,27,2,16,22,3] and performance optimization [5,11,20,19,15,33]. While these approaches have focused on a single aspect, this work tries to bridge these two by investigating how the small change of a type annotation may affect performance. This work also suggests that performance evaluation should focus on intermediate configurations as well as outlying configurations while most previous work have largely focused on outlying configurations.

7 Conclusion

Gradual typing has received a lot of attention in the past decade thanks to its promises of harmonizing static and dynamic typing. However, a systematic study of the performance landscape for gradual typing that supports fine-grained type annotations was still missing. This work solves this issue through a systematic study of around 1.25 million micro configurations that covers all type variations across the untyped-typed spectrum of parameter types.

Based on this study, we extract several major perspectives regarding gradual typing performance. First, a small change of the type annotation for a certain parameter may significantly change the performance, sometimes larger than 10 times. Second, making types more static is strongly correlated to degrading program performance. These observations indicate that, while currently treated separately, program migration and program performance should be considered in unison in future research in gradual typing. Also, better tooling support is needed for understanding, predicting, and optimizing fine-grained gradual typing to make it more practical.

Due to space limitation, several questions are left out for future investigation, such as under what context the type change for a parameter leads to more abrupt performance swings.

References

1. Allamanis, M., Barr, E.T., Ducousso, S., Gao, Z.: Typilus: neural type hints. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (jun 2020). https://doi.org/10.1145/3385412.3385997
2. Campora, J., Chen, S., Erwig, M., Walkingshaw, E.: Migrating gradual types. In: Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages. POPL '18, ACM, New York, NY, USA (2018)
3. Campora, J.P., Chen, S.: Taming type annotations in gradual typing. Proc. ACM Program. Lang. **4**(OOPSLA) (nov 2020). https://doi.org/10.1145/3428259

4. Campora, J.P., Chen, S., Walkingshaw, E.: Casts and costs: Harmonizing safety and performance in gradual typing. Proc. ACM Program. Lang. **2**(ICFP), 98:1–98:30 (Jul 2018). https://doi.org/10.1145/3236793
5. Campora, J.P., Khan, M.W., Chen, S.: Type-based gradual typing performance optimization. Proc. ACM Program. Lang. **8**(POPL) (jan 2024). https://doi.org/10.1145/3632931
6. Castagna, G., Lanvin, V., Petrucciani, T., Siek, J.G.: Gradual typing: A new perspective. Proc. ACM Program. Lang. **3**(POPL) (Jan 2019). https://doi.org/10.1145/3290329
7. Chandra, S., Gordon, C.S., Jeannin, J.B., Schlesinger, C., Sridharan, M., Tip, F., Choi, Y.: Type inference for static compilation of javascript. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 410–429. OOPSLA 2016, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2983990.2984017
8. Chaudhuri, A., Vekris, P., Goldman, S., Roch, M., Levi, G.: Fast and precise type checking for javascript. Proc. ACM Program. Lang. **1**(OOPSLA), 48:1–48:30 (Oct 2017). https://doi.org/10.1145/3133872, `http://doi.acm.org/10.1145/3133872`
9. Cristiani, F., Thiemann, P.: Generation of typescript declaration files from javascript code. In: Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. p. 97–112. MPLR 2021, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3475738.3480941
10. Feldthaus, A., Møller, A.: Checking correctness of typescript interfaces for javascript libraries. SIGPLAN Not. **49**(10), 1–16 (Oct 2014). https://doi.org/10.1145/2714064.2660215
11. Feltey, D., Greenman, B., Scholliers, C., Findler, R.B., St-Amour, V.: Collapsible contracts: Fixing a pathology of gradual typing. Proc. ACM Program. Lang. **2**(OOPSLA) (oct 2018). https://doi.org/10.1145/3276503, `https://doi.org/10.1145/3276503`
12. Greenman, B.: Gtp benchmarks for gradual typing performance. In: Proceedings of the 2023 ACM Conference on Reproducibility and Replicability. p. 102–114. ACM REP '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3589806.3600034
13. Greenman, B., Takikawa, A., New, M.S., Feltey, D., Findler, R.B., Vitek, J., Felleisen, M.: How to evaluate the performance of gradual type systems. Journal of Functional Programming **29**, e4 (2019). https://doi.org/10.1017/S0956796818000217
14. Kristensen, E.K., Møller, A.: Type test scripts for typescript testing. Proc. ACM Program. Lang. **1**(OOPSLA), 90:1–90:25 (Oct 2017). https://doi.org/10.1145/3133914, `http://doi.acm.org/10.1145/3133914`
15. Kuhlenschmidt, A., Almahallawi, D., Siek, J.G.: Toward efficient gradual typing for structural types via coercions. In: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 517–532. PLDI 2019, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3314221.3314627
16. Migeed, Z., Palsberg, J.: What is decidable about gradual types? Proc. ACM Program. Lang. **4**(POPL) (Dec 2019). https://doi.org/10.1145/3371097
17. Mir, A.M., Latoškinas, E., Proksch, S., Gousios, G.: Type4py: practical deep similarity learning-based type inference for python. In: Proceedings of the 44th International Conference on Software Engineering. pp. 2241–2252 (2022)
18. Miyazaki, Y., Sekiyama, T., Igarashi, A.: Dynamic type inference for gradual hindley–milner typing. Proc. ACM Program. Lang. **3**(POPL), 18:1–18:29 (Jan 2019). https://doi.org/10.1145/3290331
19. Moy, C., Nguy˜ên, P.C., Tobin-Hochstadt, S., Van Horn, D.: Corpse reviver: Sound and efficient gradual typing via contract verification. Proc. ACM Program. Lang. **5**(POPL) (jan 2021). https://doi.org/10.1145/3434334

20. Ortin, F., Garcia, M., McSweeney, S.: Rule-based program specialization to optimize gradually typed code. Knowledge-Based Systems **179**, 145–173 (2019). https://doi.org/https://doi.org/10.1016/j.knosys.2019.05.013
21. Peng, Y., Gao, C., Li, Z., Gao, B., Lo, D., Zhang, Q., Lyu, M.: Static inference meets deep learning: A hybrid type inference approach for python. In: Proceedings of the 44th International Conference on Software Engineering. p. 2019–2030. ICSE '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3510003.3510038
22. Phipps-Costin, L., Anderson, C.J., Greenberg, M., Guha, A.: Solver-based gradual type migration. Proc. ACM Program. Lang. **5**(OOPSLA) (oct 2021). https://doi.org/10.1145/3485488
23. Pradel, M., Gousios, G., Liu, J., Chandra, S.: Typewriter: Neural type prediction with search-based validation (2020)
24. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. pp. 481–494. POPL '12, ACM, New York, NY, USA (2012). https://doi.org/10.1145/2103656.2103714
25. Rastogi, A., Swamy, N., Fournet, C., Bierman, G.M., Vekris, P.: Safe & efficient gradual typing for typescript. In: POPL (2015)
26. Siek, J., Vitousek, M.M., Cimini, M., Tobin-Hochstadt, S., Garcia, R.: Monotonic references for efficient gradual typing (2015), `https://doi.org/10.1007/978-3-662-46669-8_18`
27. Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 Symposium on Dynamic Languages. pp. 7:1–7:12. DLS '08, ACM, New York, NY, USA (2008). https://doi.org/10.1145/1408681.1408688
28. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: LIPIcs-Leibniz International Proceedings in Informatics. vol. 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2015)
29. Takikawa, A., Feltey, D., Greenman, B., New, M.S., Vitek, J., Felleisen, M.: Is sound gradual typing dead? In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 456–468. POPL '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2837614.2837630
30. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: From scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications. pp. 964–974. OOPSLA '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1176617.1176755
31. Tunnell Wilson, P., Greenman, B., Pombrio, J., Krishnamurthi, S.: The behavior of gradual types: a user study. In: Proceedings of the 14th ACM SIGPLAN International Symposium on Dynamic Languages. p. 1–12. DLS 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3276945.3276947
32. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for python. In: Proceedings of the 10th ACM Symposium on Dynamic Languages. pp. 45–56. DLS '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2661088.2661101
33. Vitousek, M.M., Siek, J.G., Chaudhuri, A.: Optimizing and evaluating transient gradual typing. pp. 28–41. DLS 2019, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3359619.3359742
34. Vitousek, M.M., Swords, C., Siek, J.G.: Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. pp. 762–774. POPL 2017, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009849
35. Wadler, P., Findler, R.B.: Well-typed programs can't be blamed. pp. 1–16. ESOP '09, Springer-Verlag, Berlin, Heidelberg (2009), `http://dx.doi.org/10.1007/978-3-642-00590-9_1`