Type-Based Gradual Typing Performance Optimization

JOHN PETER CAMPORA, Quantinuum, USA
MOHAMMAD WAHIDUZZAMAN KHAN, University of Louisiana, USA
SHENG CHEN, University of Louisiana, USA

Gradual typing has emerged as a popular design point in programming languages, attracting significant interests from both academia and industry. Programmers in gradually typed languages are free to utilize static and dynamic typing as needed. To make such languages sound, runtime checks mediate the boundary of typed and untyped code. Unfortunately, such checks can incur significant runtime overhead on programs that heavily mix static and dynamic typing. To combat this overhead without necessitating changes to the underlying implementations of languages, we present *discriminative typing*. Discriminative typing works by optimistically inferring types for functions and implementing an optimized version of the function based on this type. To preserve safety it also implements an un-optimized version of the function based purely on the provided annotations. With two versions of each function in hand, discriminative typing translates programs so that the optimized functions are called as frequently as possible while also preserving program behaviors.

We have implemented discriminative typing in Reticulated Python and have evaluated its performance compared to guarded Reticulated Python. Our results show that discriminative typing improves the performance across 95% of tested programs, when compared to Reticulated, and achieves more than 4× speedup in more than 56% of these programs. We also compare its performance against a previous optimization approach and find that discriminative typing improved performance across 93% of tested programs, with 30% of these programs receiving speedups between 4 to 25 times. Finally, our evaluation shows that discriminative typing remarkably reduces the overhead of gradual typing on many mixed type configurations of programs.

In addition, we have implemented discriminative typing in Grift and evaluated its performance. Our evaluation demonstrations that DT significantly improves performance of Grift.

CCS Concepts: • **Theory of computation** → **Type structures**; **Program analysis**.

Additional Key Words and Phrases: gradual typing, variational types, performance optimization, type-based specialization

ACM Reference Format:
John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. 2024. Type-Based Gradual Typing Performance Optimization. *Proc. ACM Program. Lang.* 8, POPL, Article 89 (January 2024), 33 pages. https://doi.org/10.1145/3632931

1 INTRODUCTION

Dynamic and static typing are performed at different times (one at runtime and the other at compile time) but have complementary advantages, and often programmers using a language with one typing discipline desire the advantages of the other. Gradual typing has emerged as a promising approach to integrate both typing disciplines within a single language and harmonize their advantages [Siek and Taha 2006]. In a single program, programmers can modify annotations

Authors' addresses: John Peter Campora, Quantinuum, Broomfield, USA, john.campora@quantinuum.com; Mohammad Wahiduzzaman Khan, University of Louisiana, Lafayette, USA, wahid.zaman.mmu@gmail.com; Sheng Chen, University of Louisiana, Lafayette, USA, sheng.chen@louisiana.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2475-1421/2024/1-ART89
https://doi.org/10.1145/3632931

```
1  def succI(x1:Int)->Int:
2    return x1 + 1
3
4  def f1(x):
5    z = 0
6    return succI(x) + succI(z)
7
8  def f2(y):
9    return 1 if notB(y) else 3
10
11 f1 (4)
12 f2 (true)
13 f2 (5)
```

```
14 def succI(x1):
15   return x1 + 1
16
17 def f1(x):
18   z = 0
19   return succI(x : ★ ⇒^ℓx Int) + succI(z : ★ ⇒^ℓz Int)
20
21 def f2(y):
22   return 1 if notB(y : ★ ⇒^ℓy Bool) else 3
23
24 f1 (4 : Int ⇒^ℓ4 ★)
25 f2 (true : Bool ⇒^ℓtrue ★)
26 f2 (5 : Int ⇒^ℓ5 ★)
```

Fig. 1. The programs before (left) and after (right) cast insertion. We assume that notB is defined similarly as succI but has the type annotation Bool → Bool. For illustration purpose, we assume + has the type Int → Int → Int.

to use static or dynamic typing as needed. Thanks to this flexibility, gradual typing has appealed to many language designers, and it has been added to several languages, including C# [Bierman et al. 2010], Racket [Tobin-Hochstadt and Felleisen 2008], and Python [Vitousek et al. 2017].

In gradual typing, typed values can flow into untyped contexts and vice versa. A value flowing into a context may not have the type expected by that context, which will violate user-specified type invariants. If gradual typing is implemented by simply erasing type annotations then nothing prevents execution from violating the type invariants. This challenge is solved with runtime *casts* that check if values have desired types at runtime.

To illustrate, consider the Python code snippet in Figure 1 (left), where annotations are added using the syntax of Reticulated Python [Vitousek et al. 2014, 2017]. This program will be translated to the one in Figure 1 (right) before it can be run. During translation, the type annotation for succI is eliminated. Meanwhile, casts are inserted at certain locations. Casts have the form $e : G_1 \Rightarrow^{\ell_e} G_2$, meaning that, at runtime, the expression e is checked to have the type G_2. Such checks are inserted when e is known to have the type G_1 at compile time (source type) but the context requires it to have the type G_2 (target type). For example, a cast is inserted for x when it is passed into succI because its compile-time type is ★ while succI requires it to have the type Int. Each cast carries a label (for example ℓ_x), representing a program location that will be used to provide debugging information should the cast fail. Instead of giving a single label to both the function and its argument [Siek et al. 2015b], we give them different labels in the spirit of Miyazaki et al. [2019] to retain more precise blaming information. For this reason, our program variable names are as unique as possible.

1.1 Performance Problem of Gradual Typing

Casts, however, can induce runtime overhead [Herman et al. 2010; Rastogi et al. 2012]. For example, Takikawa et al. [2016] observed remarkable performance slowdowns of gradually-typed programs compared to their untyped counterparts, sometimes more than 100 times. Many methods have been developed to tackle the performance issue with gradual typing [Bauman et al. 2017; Campora et al. 2018a; Feltey et al. 2018a; Herman et al. 2010; Kuhlenschmidt et al. 2019; Muehlboeck and Tate 2017; Richards et al. 2017; Vitousek et al. 2019, 2017]. Among them, one promising approach is to eliminate redundant casts, developed by Rastogi et al. [2012] for ActionScript [Moock 2004]. We refer to this approach as IAO, since it uses type Inflows And Outflows for optimization.

To illustrate the idea of IAO, consider the cast[1] with the label ℓ_z (line 19[2]) in Figure 1. Without type inference, the variable z has the type $\star$, meaning that it will be cast when it is passed into succI. To avoid this, IAO infers the type Int for z, since it is assigned 0, and thus eliminates the cast on z since both the source and target types are Int.

A main correctness goal of IAO is that the optimization should preserve program behaviors. Specifically, optimized programs should either produce the same value, or fail and blame the same location. To realize this goal, IAO infers types for variables and function parameters for which all inflows are observable and consistent. For a variable, an inflow is an assignment to that variable. For a function, an inflow is any argument used to call that function. Roughly, IAO infers types for local variables, parameters for local functions that do not escape, and function return types. It does not infer types for parameters of non-local functions nor any variables that receive inflows from these un-optimized parameters.

While preserving program behaviors is a well-justified design decision, realizing it costs optimization opportunities in many circumstances. First, since any top-level, public functions and methods (that is they are not nested) may be used by other existing code that are unavailable for analysis while optimizing the current program, their parameter types should not be inferred. This is particularly problematic in languages without access modifiers, such as Python, where top-level functions and methods are prevalent. To illustrate, consider the function f1 in Figure 1. Based on the following two facts, it is reasonable to infer f1 to have the parameter type Int: (1) the definition of f1 requires the parameter type to be Int and (2) f1 is called only once and the type of the argument is Int. However, IAO does not infer this type to avoid violating its design goal. Specifically, if IAO infers Int and another module calls f1 with some argument that is not of the type Int, then the call fails on line 2. Instead, the expected failing site is on line 6 within f1.

Second, even if a function is private, IAO does not infer its parameter type when the function is called with values of inconsistent types. To illustrate, consider the function f2 and its two calls in Figure 1. Although it is reasonable to infer Bool as its parameter type based on the definition of f2, IAO does not infer this type for a similar reason above. Specifically, if Bool is inferred, then the call on line 13 will fail on that line. Instead, the expected failing location is on line 9.

In both f1 and f2, inferring their parameter types means that no casts will be inserted when they are translated, which leads to better performance. However, inferring these types causes the program to fail at a different location than when not inferring them. In general, identifying more optimization opportunities and preserving program behaviors are two conflicting goals.[3] When they cannot be achieved at the same time, the latter is more important, witnessed by IAO [2012]. However, in languages like Python where almost all top-level functions and methods are public, we lose too many optimization opportunities. This is unfortunate, as eliminating unnecessary casts is a promising way to address the performance problem with gradual typing [Takikawa et al. 2016].

1.2 Discriminative Typing for Optimizing Gradual Typing Performance

In this paper, we develop an approach that identifies more optimization opportunities while still preserving program behaviors. Our approach is inspired by the work from (operating) systems research, where it is common to use the idea of *fast path* for performance optimization [Huang et al. 2017; Kelsey et al. 2009; Kogan and Petrank 2012; McNamee et al. 2001; Xu et al. 2004]. Specifically, for a certain functionality, system software often implements both a *fast* version and a *slow* one.

---

[1]This cast is used to illustrate how Rastogi et al. [2012] works and is not present in Reticulated Python [Vitousek et al. 2017] since it infers that z has the type Int with its local type inference.

[2]All line numbers refer to Python source code in this paper.

[3]Greenman and Felleisen [Greenman and Felleisen 2018] explored a spectrum of type soundness and performance when different levels of soundness and blaming strategies are taken

The execution routes to the fast path if the conditions for that path are satisfied and takes the slow path otherwise.

Following that idea, our approach consists of two steps. First, for each function that we are able to optimize (that is we can infer some parameter types), we create two versions of that function during cast insertion. The fast version infers types for un-annotated parameters, and thus eliminates redundant casts inside the body of the function. The slow version, in contrast, leaves the parameter type to be dynamic and thus inserts necessary casts as usual. Second, when a function is called, the type of the argument is used to *discriminate* the version of the function to be called, thus the name *discriminative typing* (DT). Specifically, if the argument type is a subtype [Wadler and Findler 2009] (We started with the precision relation [Siek et al. 2015b] but that failed to preserve failing behaviors.) of the parameter type of the fast version of the function, then the fast version will be used to perform the function call. Otherwise, the slow version will be used.

To illustrate, consider the program in Figure 1. In the first step, we observe that we can infer the parameter types for the functions `f1` and `f2` and can thus potentially optimize them. Therefore, we have two versions for each function, as shown in Figure 2. For each function with the name `fun`, the name of the fast version is `fun_Fast` and that of the slow version is `fun`. The parameter types for `f1_Fast` and `f2_Fast` are `Int` and `Bool`, respectively. Note, we do not add these types to `f1_Fast` and `f2_Fast` explicitly in the source code but maintain them for optimization purpose during cast insertion. No casts are inserted in the bodies of the fast version of functions. The parameter types for `f1` and `f2` are dynamic ($\star$), and casts are inserted in their bodies.

In the second step, we analyze the calls to these two functions so that we can discriminate the appropriate version to use. In translating `f1(4)` (line 11), the type of the value `4` determines whether we can call `f1_Fast` or `f1`–the respective fast and slow versions of `f1`. Since `Int`, the type of `4`, is a subtype of `Int`, the parameter type of `f1_Fast`, `f1_Fast` will be used (as seen on line 34).

```
27  def f1_Fast(x):
28    z = 0
29    return succI(x) + succI(z)
30
31  def f2_Fast(y):
32    return 1 if notB(y) else 3
33
34  f1_Fast (4)
35  f2_Fast (true)
36  f2 (5 : Int ⇒^ℓ5 ★)
```

Fig. 2. The program, extending lines 14 through 22, after cast insertion by DT for the program in Figure 1 left.

We need to use `f1` when either the type of the argument (1) is statically known but is not a subtype of `Int` or (2) is the dynamic type $\star$ (statically unknown). In case (1), a type error can be statically detected but we defer it to runtime to preserve program behaviors. Going to the slow version ensures that the program can still be executed and fails at the cast ℓ_x, the expected failure location. In case (2), the type of the argument cannot be statically determined. This happens when the argument, for example, is the result of some function that may return different types (such as `eval`) or is an element of a heterogeneous data structure. This also happens when the function is called by some existing, untyped code, which happens quite often since an advantage of gradual typing is that it can work by migrating certain potions of a project to gradual typing while others remain in dynamic typing. Program behaviors are clearly preserved in case (2) since the execution goes to the slow version, which is the same as if the function call is not optimized.

Following a similar reasoning, the translation of `f2(true)` (line 12) calls the fast version `f2_Fast` (line 35). In translating `f2(5)`, since `Int`, the type of the argument, is not a subtype of `Bool`, the parameter type of `f2_Fast`, we can not used the fast version `f2_Fast`. Moreover, since `Int` is a subtype of $\star$, the parameter type of `f2`, the translation uses the slow version of `f2` (line 36).

Based on this example, we make the following observations. First, given the reasoning above, our optimization preserves program behaviors. We establish a formal account about this later. Second, our optimization indeed eliminates casts. For the program in Figure 1 left, the translated program

by IAO [Rastogi et al. 2012] or Reticulated [Vitousek et al. 2014, 2017] yields 6 casts while our approach yields 2 casts only. We present a more thorough performance evaluation in Section 6. Third, compared to standard cast-insertion methods [Siek and Taha 2006; Siek et al. 2015b], our optimization approach slightly increases the size of the translated program, but is within a factor of 2 since we generate at most two versions of each function. For the example program above, the standard translation leads to a program having 10 LOC (without blank lines) and our approach outputs a program having 15 LOC.

1.3 Contributions and Outlines of the Paper

Many methods have been developed to tackle performance issues in gradual typing, which we discuss in Section 8. In improving the performance of gradual typing, we make the following contributions.

(1) In Sections 1 and 2, we introduce the idea of discriminative typing (DT), which optimizes programs while preserving program behaviors without changes needed to the source and target languages of gradual typing. Our idea is orthogonal to most existing performance optimization approaches for gradual typing.
(2) In Section 4, we develop a method for safely providing types used in the creation of fast versions of function definitions. In order to provide an optimal type, this method must consider all of the different possibilities of inferred types for a function's parameters.
(3) In Section 5, we develop a cast insertion procedure that uses subtyping to ensure that our translated programs safely call fast versions of functions. We show that our translation preserves the execution result of the unoptimized program–either producing the same value or blaming the same location.
(4) In Section 6, we evaluate our implementation of DT across 12 benchmarks with a total of 6352 configurations. The result shows that DT achieves speedups on 95% of measured configurations (with 56% of all configurations receiving speedups of over 4 times) over guarded Reticulated Python and 93% over IAO (with 30% of all configurations receiving speedups of over 4 times). We also evaluate the performance of DT against Grift in Section 7.

The rest of the paper is organized as follows. Section 3 provides background for technical development in subsequent sections. Section 8 situates this work in the aforementioned research on improving the performance of sound gradual typing. Finally, Section 9 concludes.

2 DISCRIMINATIVE TYPING, INFORMALLY

Our optimization consists of two important ingredients: creating fast versions of functions and discriminating fast versions from slow ones when functions are used (when they are called or passed to other higher-order functions). We illustrate the challenges in them and present our solutions in Sections 2.1 and 2.2, respectively.

2.1 Finding Optimizable Parameters

For any given function, we create its slow version through normal cast insertion without inferring any parameter types. In contrast, for the fast version, we infer parameter types based on the definition of the function, allowing us to eliminate redundant casts inserted during cast insertion. We say a parameter is *optimizable* if we can assign a static type to it and that static type interacts *safely* with the rest of the function. Here safely means that if an argument satisfies that static type, then the argument will not cause execution failures when the function is called. To tap more optimization opportunities, we aim to identify as many optimizable parameters as possible.

```
37 def width(fixed,numChrs:Int,wdF,adjF):
38   if notB(fixed):
39     return wdF(fixed) + 3
40   else
41     return wdF(numChrs)+adjF(numChrs)
```

```
42 def adjust(adjF):
43   return width(true,7,f1,adjF)
44
45 adjust (f2)
46 adjust (f1)
47 adjust (notB)
```

Fig. 3. The source program for demonstrating the challenges and solutions of DT.

Identifying optimizable parameters is challenging for functions with multiple parameters. We use the function `width` from Figure 3 to illustrate the challenges. This function, adapted from Campora et al. [2018b], is for computing the width of a row when displaying a string with `numChrs` characters. The width is decided by several factors: `fixed` for determining whether the width is fixed or not, `wdF` for computing the width, and `adjF` for adjusting the width. With some manual reasoning, we realize that `adjF` is optimizable with the static type `Int` $\longrightarrow$ `Int`, `fixed` is optimizable with the type `Bool`, and `wdF` is optimizable with the type $\star \longrightarrow$ `Int`.

In the following, we first present three methods that one is likely to come up with. Unfortunately, all of them fail for some reason. After that, we present a working method.

Three non-working methods In the first method, when deciding if a parameter p is optimizable for a given function, we assign all parameters except p in that function the type $\star$. The parameter p is optimizable if it can be assigned a static type and is not otherwise. In the second method, a parameter p is optimizable if all parameters of the function can be assigned static types. Both methods fail. The first method, which is essentially the same as dynamic typing, is too permissive. For example, this method will infer `wdF` as having the type `Int` $\longrightarrow$ `Int` while in fact it should be $\star \longrightarrow$ `Int`. The second method, which is essentially static typing, is too restrictive.

In the third method, we try to circumvent the problems with the first two methods by assigning static types to a set of relevant parameters (*RP*) when deciding whether a parameter p is optimizable. Note that *RP* includes at least p. Once we have *RP*, p is optimizable if all parameters in *RP* can be given static types and is not otherwise. For collecting *RP*, one idea is to let *RP* be the set of parameters that share typing constraints, directly or indirectly, with p.

However, a particular perplexity of this method is the asymmetry of the results about optimizable parameters. Specifically, if two parameters have the same *RP*, then we expect that either both are optimizable or neither is optimizable. However, this expectation does not hold when we consider the parameters `fixed` and `wdF` in the function `width`. Clearly, the *RP* for them is identical, but we previously observed that `fixed` is optimizable with a fully static type while `wdF` is optimizable with a partially static type (We can not infer a partial type for a parameter based on Garcia et al. [2015]). A potential reason for this asymmetry is that `fixed` and `wdF` *conditionally* interact in the then-branch of the if-expression. To accommodate for this asymmetry, we need to collect conditionally relevant parameters, making the implementation of this method challenging.

A working method Instead of trying to find a correct way to compute *RP* in the third method above, we explore a different method based on the definition of optimizable. Before continuing, we introduce a term that will be used throughout the paper. We use the term *configuration* to refer to an assignment of a type (either a $\star$ or the best possible static type, which can be computed through type inference) to each dynamic parameter (whose annotation is a $\star$) of a function. An example configuration for the function `width` assigns `Bool` to `fixed` and $\star$s to `wdF` and `adjF`.

A parameter p for a statically well-typed gradual program is optimizable if it has a static type and the best static type T interacts safely with the rest of the function. The interaction is guaranteed to be safe if the type correctness of the function under all possible configurations is preserved, regardless

of whether p has the type $\star$ or T. Specifically, let us consider two cases for each configuration: (1) the function is always ill typed and (2) the function is always well typed. In case (1), the type error is not caused by p because the type error already exists even when p has the type $\star$ and $\star$ cannot cause type errors. In case (2), p will not cause a type error because the function is still well typed when p has the type T. Here we consider all possible configurations because identifying relevant configurations to consider is difficult, as shown by the non-working methods earlier.

Unfortunately, a brute-force implementation of this idea is infeasible, as the number of configurations to type check is exponential in the number of the parameters. We instead employ the idea of variational typing [Chen et al. 2012, 2014] to reuse the typing process to reduce complexity. Specifically, for each parameter with a gradual type, we assign a variational type that encodes two possible types: the type given in the annotation by the user and a most static type that can be given to the parameter without causing type errors (this most static type could be found through type inference). As such, instead of running the type checking process an exponential time, we run it just once with variational typing. We will give details about the typing time in Section 6. After the typing finishes, we collect all parameters whose best static type assignments do not change the type correctness of all possible configurations of the function as optimizable parameters, a condition identified earlier. This allows us to identify `fixed` (with parameter type `Bool`), `adjF` (with type `Int` $\rightarrow$ `Int`), and `wdF` (with type $\star \rightarrow$ `Int`) as optimizable. For `adjust`, the parameter `adjF` (with type `Int` $\rightarrow$ `Int`) is optimizable.

2.2 Discriminating Fast From Slow Versions

After computing the types for fast and slow versions of functions in the first step, the second step translates gradual programs into target programs. DT has two main goals for translation. First, it aims to translate as many calls into using the fast versions as possible since they contain fewer casts than their corresponding slow versions. Second, the translation should preserve program execution behaviors. Specifically, optimized programs either produce the same value or fail and blame the same location as unoptimized programs [Siek et al. 2015b; Wadler and Findler 2009].

To realize these goals, we need to correctly discriminate when it is safe to use fast versions of functions throughout the program. In the introduction, we discussed using the subtyping relation for discriminating functions. Specifically, if the type of the argument is a subtype of the parameter type of the fast version, then the fast version is used; otherwise, the slow version is used. The subtyping relation mainly includes the following: (1) a base type is a subtype of itself, (2) $\star$ is a subtype of itself, (3) function subtyping is contravariant for domains and covariant for ranges, and (4) a type is a subtype of $\star$ if it is a subtype of some base type or $\star \rightarrow \star$ (subtyping factors through ground types). We give a function to compute the subtyping relation in Section 5.3.

Fortunately, that relation also works for higher-order functions. To illustrate, consider the function call `adjust (f2)` (line 45), where we need to discriminate not only the fast and slow versions of `adjust` but also those of `f2`. Thus, we need to consider four possibilities of translating the function call. Without optimization, this function call leads to a runtime error that blames `y`–failing on line 9 because the variable `y` does not have the type `Bool` at runtime.

We translate line 45 as follows. First, since `Bool` $\rightarrow$ `Int`, the type of `f2_Fast`, is not a subtype of `Int` $\rightarrow$ `Int`, the parameter type of `adjust_Fast`, we cannot translate the call to `adjust_Fast(f2_Fast)`. Similarly, since `Bool` $\rightarrow$ `Int` is not a subtype of $\star$, the call cannot be translated to `adjust(f2_Fast)`. Next, since $\star \rightarrow$ `Int`, the type of `f2` is a subtype of `Int` $\rightarrow$ `Int`, the parameter type of `adjust_Fast`, we can translate the function call to `adjust_Fast(f2)`. Indeed, this optimized expression preserves

failure behaviors, as shown in the following reduction sequence, failing and blaming `y` inside `f2`.

$$\begin{aligned}
& \texttt{adjust_Fast (f2} : \star \to \texttt{Int} \Rightarrow^{\ell_{f2}} \texttt{Int} \to \texttt{Int)} \\
\longrightarrow\ & \texttt{width_Fast(true,7,f1} : \star \to \texttt{Int} \Rightarrow^{\ell_{f1}} \star \to \texttt{Int,f2} : \star \to \texttt{Int} \Rightarrow^{\ell_{f2}} \texttt{Int} \to \texttt{Int)} \\
\longrightarrow^*\ & \cdots \texttt{+ (f2} : \star \to \texttt{Int} \Rightarrow^{\ell_{f2}} \texttt{Int} \to \texttt{Int) (7)} \longrightarrow \texttt{f2(7} : \texttt{Int} \Rightarrow^{\overline{\ell_{f2}}} \star\texttt{)} : \texttt{Int} \Rightarrow^{\ell_{f2}} \texttt{Int} \\
\longrightarrow^*\ & \cdots \texttt{ notB((7} : \texttt{Int} \Rightarrow^{\overline{\ell_{f2}}} \star\texttt{)} : \star \Rightarrow^{\ell_y} \texttt{Bool)} \longrightarrow^* \texttt{blame } \ell_y
\end{aligned}$$

We can similarly translate the function call on line 46 into `adjust_Fast(f1_Fast)` and that on line 47 into `adjust (notB` : $\texttt{Bool} \to \texttt{Bool} \Rightarrow^{\ell_n} \star$`)`. We present the translation output in Appendix C of the companion report [Campora et al. 2023]. We can verify that all translations preserve program behaviors. We formally present a translation method and prove its correctness in Section 5.

3 BACKGROUND

This section introduces necessary background on variational typing for technical developments in Sections 4 and 5. Since we have introduced gradual typing while we were discussing various examples in previous sections, we do not present it separately in this section.

A variational program is a compact representation [Erwig and Walkingshaw 2011] of a large family of different but related programs, such as software product lines [Apel et al. 2016]. Variational programs are obtained by introducing named *variations* into programs. Each variation contains two alternatives: *first* and *second* (sometimes called *left* and *right*). For example, the following expression `vpA1` contains a variation named A, which has the first alternative `f1` and second alternative `f1_Fast`. The functions `f1` and `f1_Fast` were introduced in Section 1.

```
48  vpA1 = A⟨f1,f1_Fast⟩ (3)
49  vpA2 = A⟨f1,f1_Fast⟩ (B⟨3,True⟩)
```

A *selection* process takes in a variational expression e and a *selector* having the form $d.i$ and replaces all variations named d in e with their ith alternative. The notation of selection is $\lfloor e \rfloor_{d.i}$. For example, $\lfloor \texttt{vpA1} \rfloor_{A.1}$ yields `f1 (3)`, $\lfloor \texttt{vpA1} \rfloor_{A.2}$ yields `f1_Fast (3)`, and $\lfloor \texttt{vpA2} \rfloor_{B.1}$ yields `vpA1` (line 48). We call selectors of the form $d.1$ ($d.2$) left (right) selectors. A decision is a set of selectors, and we use δ to range over decisions. Selection extends naturally to decisions by recursively selecting with each selector in the decision. For example, $\lfloor \texttt{vpA2} \rfloor_{\{A.1,B.2\}}$ leads to `f1 (True)`. A variant or a plain program can be obtained from a variational program by eliminating all variations in it.

The goal of variational typing is to assign types to variational expressions [Chen et al. 2012, 2014]. One challenge in variational typing is that the standard approaches to typing expressions are too strict. To illustrate, consider typing the expression `vpA1`. The function $A\langle\texttt{f1},\texttt{f1_Fast}\rangle$ has the type $A\langle\star \to \texttt{Int}, \texttt{Int} \to \texttt{Int}\rangle$, which is not a function type. As a result, standard typing rules do not apply. On the other hand, both `f1 (3)` and `f1_Fast (3)`, encoded in `vpA1`, are well typed.

To address this issue, variational typing introduced a type equivalence relation and allowed a type to be transformed to a new type that is equivalent to it. Intuitively, a type M_1 is equivalent to another type M_2, written as $M_1 \equiv M_2$, if selecting them with any decision yields the same type. For example, $A\langle\star \to \texttt{Int}, \texttt{Int} \to \texttt{Int}\rangle \equiv A\langle\star, \texttt{Int}\rangle \to \texttt{Int}$ because selecting them with $A.1$ always leads to $\star \to \texttt{Int}$ and selecting with $A.2$ always yields $\texttt{Int} \to \texttt{Int}$. Migrational typing [Campora et al. 2018b] extended the equivalence relation to work with gradual types so that two variational types are consistent if selecting them with any decision leads to consistent types. As a result, `vpA1` has the type `Int` since the type of its function is equivalent to $A\langle\star, \texttt{Int}\rangle \to \texttt{Int}$, whose parameter type is consistent with `Int`, the type of the argument `3`.

What about the expression `vpA2`? According to the reasoning above, the parameter type $A\langle\star, \texttt{Int}\rangle$ is not consistent with $B\langle\texttt{Int}, \texttt{Bool}\rangle$, the type of the argument. As a result, we cannot assign a type to `vpA2`. It is quite common that some variants in a variational program are ill typed while the majority

of variants are well typed. For example, certain configurations of a gradual program may be ill typed, and the variational program that encodes all configurations of the gradual program contains ill-typed variants. Therefore, assigning types to such variational programs is highly desirable.

Error-tolerant typing [Chen et al. 2012] and migrational typing [Campora et al. 2018b] addressed this issue by extending the typing process with a *typing pattern*. A typing pattern can be a $\bot$, a $\top$, or a variation between two typing patterns. The typing pattern specifies the validity of the typing for each variant encoded in typing the variational program. If the pattern for a variant is $\top$, then the typing is valid, and the result can be made use of later. If the pattern is $\bot$, then the typing for that variant is invalid, and the result should be discarded. For example, if our goal was to find the configuration that assigns static types to as many parameters as possible, we should consider those configurations whose typing pattern are $\top$s and ignore those whose typing patterns are $\bot$s. With this idea, the expression `vpA2` can still be given the type `Int` but with the typing pattern $A\langle\top, B\langle\top,\bot\rangle\rangle$, meaning that the type `Int` for the variant $A.1$ is valid and that for the variant $A.2$ is invalid. With this extension, we can assign a type to any variational program.

The work by Campora et al. [2018b] also uses variational types to find most static types for parameters in gradual typing. However, that work searches type assignments that do not cause *static type errors* while here we search for type assignments that do not introduce *runtime type errors*. For example, for `width`, one best migration identified by that work is adding type annotations to `wdF` (with the type `Int` $\longrightarrow$ `Int`) and `adjF` but not `fixed`. However, we know that assigning `Int` $\longrightarrow$ `Int` to `wdF` may lead to a runtime error and we find the type $\star \longrightarrow$ `Int` for `wdF`. Also, unlike this work, that work does not translate programs for optimizing performance.

4 FINDING OPTIMIZABLE PARAMETERS AND FUNCTION TYPES

Our optimization consists of two steps. In the first step (this section), we compute the types of slow and fast versions of functions. In the second step (Section 5), we perform translation using the type information from the first step. The main challenge in this step is finding optimizable parameters. To find all of them, we first type check all possible configurations of a function through variational typing (Section 4.1) and then interpret the typing results to find optimizable parameters (Section 4.2). After that, we calculate types for slow and fast versions of functions in Section 4.3.

4.1 Typing All Configurations

The syntax of the source language we consider and that for relevant types and environments are presented in the upper part of Figure 4. We have five categories of types, and they are related as follows. Base types (U) are non-function constant types, static types (T) extend them with function types, and gradual types (G) extend static types with a $\star$. Variational types (V) and variational gradual types (M) extend static types and gradual types with variations, respectively. The variation names C and C_is are used specifically for representing expressions whose types are different in conditional branches. We discuss them in detail when we discuss the rule for typing conditionals. The choice environment (Ω) records the variational types we assigned to parameters. This information is used in determining what static types can we give to optimizable parameters.

We present the typing rules in the bottom part of Figure 4. Our typing judgments have the form $\pi;\Gamma \vdash e : M \mid \Omega$, meaning that under Γ, the types of all configurations of e are encoded in M with their validity specified by π, and the static types for dynamic parameters under all configurations are encoded in Ω. The intuition of using π to specify typing validity was given in Section 3. For the purpose of typing, we transform the expression **fun** $f\ x_1 : G_1, x_2 : G_2, \ldots, x_n : G_n = e$ to $\lambda x_1 : G_1.\lambda x_2 : G_2.\lambda \ldots \lambda x_n : G_n.e$.

The variations are introduced when typing abstractions, through the relation $\multimap$, which is also defined in Figure 4. Intuitively, $G \multimap M$ if each $\star$ in G is replaced with a new variation with the

Term variables x, y, z Value constants c Choice names A, C, d

$$
\begin{array}{llcl}
\text{Expressions} & e & ::= & c \mid x \mid \textbf{fun}\ f\ \overline{x : G} = e \mid e\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
\text{Base types} & U & ::= & \texttt{Bool} \mid \texttt{Int} \\
\text{Static types} & T & ::= & U \mid T \to T \\
\text{Gradual types} & G & ::= & U \mid G \to G \mid \star \\
\text{Variational types} & V & ::= & U \mid V \to V \mid C\langle V, V\rangle \mid d\langle V, V\rangle \\
\text{Variational gradual types} & M & ::= & U \mid M \to M \mid \star \mid C\langle M, M\rangle \mid d\langle M, M\rangle \\
\text{Typing patterns} & \pi & ::= & \top \mid \bot \mid d\langle \pi, \pi\rangle \\
\text{Type environment} & \Gamma & ::= & \varnothing \mid \Gamma, x \mapsto M \\
\text{Choice environment} & \Omega & ::= & \varnothing \mid \Omega, x \mapsto M
\end{array}
$$

$$
\textsc{Con}\ \frac{c \text{ is of type } U}{\pi; \Gamma \vdash c : U \mid \varnothing}
\qquad
\textsc{Abs}\ \frac{G \multimap M \qquad \pi; \Gamma, x \mapsto M \vdash e : M_1 \mid \Omega}{\pi; \Gamma \vdash \lambda x : G.e : M \to M_1 \mid \Omega \cup \{x \mapsto M\}}
\qquad
\textsc{Var}\ \frac{x \mapsto M \in \Gamma}{\pi; \Gamma \vdash x : M \mid \varnothing}
$$

$$
\textsc{App}\ \frac{\pi; \Gamma \vdash e_1 : M_1 \mid \Omega_1 \qquad \pi; \Gamma \vdash e_2 : M_2 \mid \Omega_2 \qquad M_1 \approx_\pi M_2 \to M_3}{\pi; \Gamma \vdash e_1\ e_2 : M_3 \mid \Omega_1 \cup \Omega_2}
$$

$$
\textsc{If}\ \frac{\begin{array}{c} C_i\ \mathit{fresh} \qquad C_i\langle \pi_2, \pi_3\rangle; C_i\langle \Gamma_2, \Gamma_3\rangle \vdash e_1 : M_1 \mid \Omega_1 \\ M_1 \approx_{C_i\langle \pi_2, \pi_3\rangle} \texttt{Bool} \qquad \pi_2; \Gamma_2 \vdash e_2 : M_2 \mid \Omega_2 \qquad \pi_3; \Gamma_3 \vdash e_3 : M_3 \mid \Omega_3 \qquad \Omega = \Omega_1 \cup \Omega_2 \cup \Omega_3 \end{array}}{C_i\langle \pi_2, \pi_3\rangle; C_i\langle \Gamma_2, \Gamma_3\rangle \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : C_i\langle M_2, M_3\rangle \mid \Omega}
$$

$$
U \multimap U
\qquad
\frac{d\ \mathit{fresh}}{\star \multimap d\langle \star, V\rangle}
\qquad
\frac{G_1 \multimap M_1 \qquad G_2 \multimap M_2}{G_1 \to G_2 \multimap M_1 \to M_2}
$$

$$
G \sim G \qquad \star \sim G \qquad G \sim \star \qquad \frac{G_1 \multimap G_3 \qquad G_2 \multimap G_4}{G_1 \to G_2 \multimap G_3 \to G_4}
$$

$$
M_1 \approx_\pi M_2 \text{ is defined as } \forall \delta. \lfloor \pi \rfloor_\delta = \top \Rightarrow \lfloor M_1 \rfloor_\delta \sim \lfloor M_2 \rfloor_\delta
$$

Fig. 4. Syntax, typing rules, and auxiliary relations. Please see the second paragraph of Section 4.1 on how to convert the **fun** construct into a lambda form.

fresh name d. Moreover, the first alternative of the new variation is a $\star$ and the second alternative is any static variational type that makes the type checking of the whole expression as well-typed as possible. The definition of $\multimap$ consists of three rules. The first rule deals with base types U. Since U does not contain any $\star$, we have $U \multimap U$. The second rule handles $\star$. This rule formalizes the intuition we gave above. We observe that the second alternative (M) of the variation d is unconstrained. As such, we may use any M, reminiscent of the type for the parameter in typing abstractions in implicitly-typed lambda calculus. During type inference, the second alternative will be a fresh type variable such that it can be constrained to a static type due to typing constraints. We discuss more about type inference at the end of this subsection. The third rule deals with function types.

The $\multimap$ relation allows us to handle $\star$s, static types, and partially static types uniformly. The variation introduction is recorded in Ω. For simplicity, we assume bound variable names are unique.

When typing constants and variable references, we do not restrict the typing pattern π. The reason is that the typing results for them are always valid, and we can thus use any π. The Ω in both cases are $\varnothing$ because no new variations are introduced when typing them. In typing applications

(App), we use the relation $M_1 \approx_\pi M_2 \rightarrow M_3$ (defined in Figure 4) to specify that M_1 is consistent ($\sim$) with $M_2 \rightarrow M_3$ in every variant where π is $\top$. Intuitively, this means that we only require that the parameter type and the type of the argument be consistent where the typing result must be valid (π has a $\top$). For variants where π has $\perp$s, we do not place any consistency requirement about the parameter type and the type of the argument since the typing result in these variants will be discarded when we interpret the typing results to get optimizable parameters. The definition of consistency is standard in gradual typing [Siek and Taha 2006] and is given in Figure 4. When typing applications, the Ωs for subexpressions have disjoint domains.

A main subtlety of our type system happens in typing conditionals. At first, it may sound reasonable to require that conditional branches to have the same type, and so do variables used in both branches. However, applying this requirement to our type system will cause superficial typing constraints that do not exist at runtime. To illustrate, consider the function `width` from Figure 3. In a static type system, calling `wdF` with `numChrs` (in the else branch) requires the parameter type of `wdF` to be `Int`, the type annotation of `numChrs`. The type information for `wdF` is propagated to the then branch, requiring `fixed` to have the type `Int` as it is passed to `wdF` (in the then branch). Unfortunately, in the condition, `fixed` is required to have the type `Bool`, leading to a type inconsistency on `fixed` and making us incorrectly conclude that `fixed` is not optimizable.

To address this issue, we should prevent this information propagation across branches since at runtime only one branch can be taken. We fulfill that prevention by introducing *conditional* types. For example, the conditional type $C\langle \texttt{Bool} \rightarrow \texttt{Int}, \texttt{Int} \rightarrow \texttt{Int}\rangle$, where C is fresh, indicates that the type is `Bool` $\rightarrow$ `Int` in the then branch and is `Int` $\rightarrow$ `Int` in the else branch. If we assign this type to `wdF`, the requirement that `fixed` should have the type `Int` disappears. We can thus infer that `fixed` is optimizable and has the type `Bool`.

The rule If types conditionals under this idea. The rule allows branch types to be different (M_2 and M_3) and the same variable in different branches to have different types through distinct branch type environments (Γ_2 and Γ_3). They (Γ_2 and Γ_3) are merged under the the fresh variation to type check the condition. If $\Gamma_2\ (x) = M_l$ and $\Gamma_3\ (x) = M_r$, then $C_i\langle \Gamma_2, \Gamma_3\rangle(x) = C_i\langle M_l, M_r\rangle$. If a binding appears in just one environment, then the merged environment contains that binding. The premise $M_1 \approx_{C_i\langle \pi_2, \pi_3\rangle} \texttt{Bool}$ specifies that M_1 is required to be consistent with `Bool` in variants where $C_i\langle \pi_2, \pi_3\rangle$ has $\top$s only. This makes sense because only at those variants are the typings of the conditional valid (please refer to error-tolerant typing in Section 3 for more information).

With If, we can assign the type $C\langle \texttt{Int}, \texttt{Int}\rangle$, which is the same as `Int`, to the conditional in `width` with the pattern $\top$ under the following type environment.

$$\Gamma_w = \Omega_w = \{\texttt{fixed} \mapsto A\langle \star, \texttt{Bool}\rangle, \texttt{wdF} \mapsto B\langle \star, C\langle \texttt{Bool} \rightarrow \texttt{Int}, \texttt{Int} \rightarrow \texttt{Int}\rangle\rangle, \texttt{adjF} \mapsto D\langle \star, \texttt{Int} \rightarrow \texttt{Int}\rangle\}$$

where A, B, and D are the variations introduced for the parameters `fixed`, `wdF`, and `adjF`, respectively. Similarly, we can type the function `width` with a pattern $\pi_w = \top$, meaning that all configurations are well typed. The choice environment Ω_w for `width` is exactly the same as Γ_w.

In the rule Abs, the relation $\multimap$ is used to introduce fresh variations, and the second alternative of each variation is a variational type V. In implementation, a fresh type variable is used, which will be instantiated to have a V after type inference finishes. We present a type inference algorithm in Appendix E of the companion paper [Campora et al. 2023].

4.2 Finding Optimizable Parameters and Their Types

After typing the expression, we have π_f and Ω_f, from which we describe how to find optimizable parameters and their types. For finding optimizable parameters, we only need its typing pattern π_f. To begin with, we define a function $chcs\,(\Omega_f)$ for returning the set of variation names that are not C_is in Ω_f. For example, $chcs\,(\Omega_w) = \{A, B, D\}$. $chcs\,(\cdot)$ extends naturally to expressions and types.

To simplify our discussion, we assume π_f is *normalized* in the sense that it does not contain a variation for which both alternatives are equivalent (Section 3) and any path from a leaf to the root contains each unique variation just once. For example, let $\pi_5 = A\langle\top, B\langle\top,\bot\rangle\rangle$, then π_5 is normalized. In contrast, $A\langle\top,\top\rangle$ is not, but its equivalent pattern $\top$ is. Similarly, $A\langle\bot, A\langle\bot,\top\rangle\rangle$ is not normalized, but its equivalent pattern $A\langle\bot,\top\rangle$ is. A pattern can be normalized in quadratic time [Chen et al. 2014]. Also, we assume π_f is not a $\bot$ as this indicates that f contains a static type error. We further assume that π_f does not contain C variations and lift this assumption later.

Intuitively, according to Section 2.1, a parameter is optimizable if assigning it with a static type does not affect the type correctness of the function. Technically, a parameter corresponding to the variation d is optimizable if $\lfloor\pi_f\rfloor_{\{d.1\}\cup\delta} = \lfloor\pi_f\rfloor_{\{d.2\}\cup\delta}$, where δ is any decision that does not include the left or right selector of d. However, an algorithm directly based on this condition is complex since we need to consider all possible δs.

Surprisingly, the condition above inspires a very simple approach for computing optimizable parameters. Specifically, if d satisfies $\lfloor\pi_f\rfloor_{\{d.1\}\cup\delta} = \lfloor\pi_f\rfloor_{\{d.2\}\cup\delta}$ for all δs, then both alternatives of d must be the same. Conversely, this result also holds, as expressed in the following theorem. We provide its proof in Appendix B of the long version of this paper.

Theorem 4.1. *$d \in chcs(\pi_f)$ implies that $\lfloor\pi_f\rfloor_{\{d.1\}\cup\delta} \neq \lfloor\pi_f\rfloor_{\{d.2\}\cup\delta}$ for some δ.*

Based on this theorem, a parameter is optimizable if its corresponding variation does not appear in π_f. Since π_w is $\top$, then $chcs(\pi_w)$ is $\varnothing$, and all parameters of `width` are optimizable.

If the pattern π_f contains a C variation, then we apply this process to each alternative of C. If the pattern for one branch is $\bot$, then all parameters used in that branch are not optimizable. After that, we can take the intersection of the optimizable parameter sets for the alternatives as the overall optimizable parameters. This idea generalizes if π_f contains multiple C variations.

4.3 Types for Fast and Slow Versions of Functions

We now show how to derive types for slow and fast versions for any function f given its optimizable parameters, the choice environment Ω_f, and its result type M_f.

Given the variational type M_f of the function f, the type of the slow version of f should replace all non-C variations in M_f with their first alternatives. Based on the rule Abs (Section 4.1), the first alternatives are always $\star$s, meaning that the types for dynamic parameters remain $\star$s.

For the fast version of the function, we need to distinguish between optimizable and non-optimizable parameters. For optimizable parameters, we find their types from Ω_f. For other parameters, we use $\star$s. For each optimizable parameter, Ω_f specifies its static type that it should have. Specifically, if x is optimizable, the second alternative of the variational type Ω_f (x) is the type for x. If the second alternative is a C variation $C\langle T_1, T_2\rangle$, then we need to merge the two alternatives with $T_1 \sqcup T_2$. The operation $\sqcup$ is defined as follows.

$$U \sqcup U = U \quad U_1 \sqcup U_2 = \star \quad U_{11} \to U_{12} \sqcup U_{21} \to U_{22} = U_{11} \sqcup U_{21} \to U_{12} \sqcup U_{22}$$

For example, from Ω_w, the static type for `fixed` is `Bool` and that for `adjF` is `Int` $\to$ `Int`. It is possible that the return type of the function still contains variations. For such variations, we take their first alternatives. We give the rationale behind this decision in Section 5.4.

For example, for `width`, the optimizable parameters are `fixed`, `wdF`, and `adjF` (Section 4.2), the choice environment is Ω_w (Section 4.1), and its type is $A\langle\star, \texttt{Bool}\rangle \to \texttt{Int} \to B\langle\star, C\langle\texttt{Bool} \to \texttt{Int}, \texttt{Int} \to \texttt{Int}\rangle\rangle \to D\langle\star, \texttt{Int} \to \texttt{Int}\rangle \to \texttt{Int}$. Following the idea above, the type of the slow version of `width` is $\star \to \texttt{Int} \to \star \to \star \to \texttt{Int}$ and that of the fast version is $\texttt{Bool} \to \texttt{Int} \to (\star \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$. Note, here $\star \to \texttt{Int}$ is obtained from $\texttt{Bool} \to \texttt{Int} \sqcup \texttt{Int} \to \texttt{Int}$.

After obtaining the types for functions, we add them into the initial environment. To make the distinction clear, we use $\ddot{\Gamma}$ to denote this new environment. For each function that has both slow and fast versions, we create a unique H variation (usually the variation name has a subscript that is the name of the function or its first letter) whose first and second alternatives are the types for the slow and fast versions of the function, respectively. We add the pair of function name and the newly created variation to $\ddot{\Gamma}$. Note, these variations have nothing to do with the variations created for typing all configurations in Section 4.1, and we use variations Hs to make this distinction clear. The variations introduced here are to facilitate cast insertion process, which we elaborate in Section 5. For example, we add `adjust` $\mapsto H_a\langle\star \to \texttt{Int}, (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\rangle$, `f1` $\mapsto H_{f1}\langle\star \to \texttt{Int}, \texttt{Int} \to \texttt{Int}\rangle$, and `f2` $\mapsto H_{f2}\langle\star \to \texttt{Int}, \texttt{Bool} \to \texttt{Int}\rangle$ to $\ddot{\Gamma}$. For each function that has only one version, we add the function name and the type to $\ddot{\Gamma}$ without creating a variation. For example, for `succI`, we add $(\texttt{succI}, \texttt{Int} \to \texttt{Int})$ to $\ddot{\Gamma}$.

5 CAST INSERTION THROUGH DISCRIMINATIVE TYPING

In this section, we present a cast insertion strategy that uses types to discriminate fast versions of functions from slow versions. We also investigate the properties of the cast insertion strategy. The syntax of target expressions, which is fully standard, is presented in Figure 5. Note, before translating a program, the type information for all of its functions have been added to $\ddot{\Gamma}$. As a result, during the translation process, we can access types of functions from $\ddot{\Gamma}$.

5.1 Translation Overview

Before presenting the translation rules, we make an important observation that drives the design (and also implementation) of the translation. Specifically, we observe that, during translation, we cannot decide whether we should use the fast version or slow version of an expression when we first encounter it. Instead, such a decision must be deferred until all arguments are seen. For example, when translating `adjust(f2)` (line 45), we are not sure whether we should translate `adjust` to `adjust` or `adjust_Fast` when we first see `adjust`, and do not know until observing the argument `f2`. When the function call has multiple arguments, the decisions on earlier arguments cannot be made until the last argument is supplied.

A natural way to support this deferred decision is by breaking the translation into two phases: the *expanding* phase that builds up possible translations as the function and earlier arguments are seen and the *refining* phase that chooses an appropriate translation from all the accumulated translations. Variations are particularly suited to formalizing this process. Specifically, we introduce variations at the expanding phase and find a translation that uses the fast versions of as many functions as possible at the refining phase.

In addition, using variations has two other advantages. First, each function is likely to have two types, and variations naturally express types for expressions that can have multiple types. Second, variations help improve the efficiency of cast insertion. For a function call with n arguments, we need to potentially explore 2^{n+1} translations of that call since we may have two versions for the function and each argument. These explorations share computations, and we can use variational typing to realize that.

We present the translation rules in Figure 5. The translation judgment has the form $\ddot{\Gamma} \vdash_I e \rightsquigarrow e_t : M$, meaning that under the type environment $\ddot{\Gamma}$, the source expression e is translated to the target expression e_t, and e_t has the type M. Note, the translated result from $\vdash_I$ may still include variations in casts and types due to the expanding phase mentioned above. All these variations will be eliminated through application rule App and the translation relation $\vdash_M$, which is defined in Figure 5.

$$e_t ::= c \mid x \mid \textbf{fun}\ f\ \overline{x} = e_t \mid e_{1t}\ (\overline{e_{2t}}) \mid \textbf{if}\ e_t\ \textbf{then}\ e_t\ \textbf{else}\ e_t \mid d\langle e_t, e_t\rangle \mid e_t : M_1 \Rightarrow^{\ell} M_2$$

$$\textsc{Con}\ \frac{c \text{ is of type } U}{\ddot{\Gamma} \vdash_I c \rightsquigarrow c : U} \qquad \textsc{Var-U}\ \frac{\ddot{\Gamma}\,(x) = G}{\ddot{\Gamma} \vdash_I x \rightsquigarrow x : G} \qquad \textsc{Var-B}\ \frac{\ddot{\Gamma}\,(x) = H_i\langle G_1, G_2\rangle}{\ddot{\Gamma} \vdash_I x \rightsquigarrow H_i\langle x, \texttt{x_Fast}\rangle : H_i\langle G_1, G_2\rangle}$$

$$\textsc{Abs-U}\ \frac{\ddot{\Gamma}\,(f) = \overline{G_1} \to G_3 \qquad \ddot{\Gamma}, \overline{x : G_1} \vdash_M e \rightsquigarrow e_t : G_3}{\ddot{\Gamma} \vdash_I \textbf{fun}\ f\ \overline{x : G_1} = e \rightsquigarrow \textbf{fun}\ f\ \overline{x} = e_t : \overline{G_1} \to G_3}$$

$$\textsc{Abs-B}\ \frac{\ddot{\Gamma}\,(f) = H_i\langle \overline{G_1} \to G_3, \overline{G_2} \to G_4\rangle \qquad \ddot{\Gamma}, \overline{x : G_1} \vdash_M e \rightsquigarrow e_{1t} : G_3 \qquad \ddot{\Gamma}, \overline{x : G_2} \vdash_M e \rightsquigarrow e_{2t} : G_4}{\ddot{\Gamma} \vdash_I \textbf{fun}\ f\ \overline{x : G_1} = e \rightsquigarrow H_i\langle \textbf{fun}\ f\ \overline{x} = e_{1t}, \textbf{fun}\ f\texttt{_Fast}\ \overline{x} = e_{2t}\rangle : H_i\langle \overline{G_1} \to G_3, \overline{G_2} \to G_4\rangle}$$

$$\textsc{If}\ \frac{\begin{array}{c}\ddot{\Gamma} \vdash_I e_1 \rightsquigarrow e_{1t} : M_1 \qquad \ddot{\Gamma} \vdash_I e_2 \rightsquigarrow e_{2t} : M_2 \qquad \ddot{\Gamma} \vdash_I e_3 \rightsquigarrow e_{3t} : M_3 \\ M = M_2 \sqcup M_3 \qquad e'_{1t} = \llbracket e_{1t} : M_1 \Rightarrow^{\ell_1} \texttt{Bool} \rrbracket\end{array}}{\ddot{\Gamma} \vdash_I \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \rightsquigarrow \textbf{if}\ e'_{1t}\ \textbf{then}\ \llbracket e_{2t} : M_2 \Rightarrow^{\ell_2} M \rrbracket\ \textbf{else}\ \llbracket e_{3t} : M_3 \Rightarrow^{\ell_3} M \rrbracket : M}$$

$$\textsc{App}\ \frac{\begin{array}{c}\ddot{\Gamma} \vdash_I e_1 \rightsquigarrow e_{1t} : M \qquad \overline{\ddot{\Gamma} \vdash_I e_2 \rightsquigarrow e_{2t} : M_2}^{\,n} \\ (\overline{M_1}, M_3, \pi_s) = split\,(M, n) \qquad \pi = \otimes\, \overline{M_2 \lessdot M_1} \otimes \pi_s \qquad \delta = bestDesc\,(\pi, chcs\,(e_{1t}\ (\overline{e_{2t}}))) \\ \delta_1 = escDesc\,(\lfloor M_3 \rfloor_\delta, \delta, chcs\,(e_{1t}\ (\overline{e_{2t}})))\end{array}}{\ddot{\Gamma} \vdash_I e_1\ \overline{e_2} \rightsquigarrow \lfloor \llbracket e_{1t} : M \Rightarrow^{\ell_1} \overline{M_1} \to M_3 \rrbracket\ (\overline{\llbracket e_{2t} : M_2 \Rightarrow^{\ell_2} M_1 \rrbracket}) \rfloor_{\delta_1} : \lfloor M_3 \rfloor_{\delta_1}}$$

$$\textsc{Main}\ \frac{\ddot{\Gamma} \vdash_I e \rightsquigarrow e_t : M \qquad \delta = allFst\,(chcs\,(e_t))}{\ddot{\Gamma} \vdash_M e \rightsquigarrow \lfloor e_t \rfloor_\delta : \lfloor M \rfloor_\delta}$$

$$\llbracket e_t : M \Rightarrow^{\ell} M \rrbracket = e_t \qquad \llbracket e_t : M_1 \Rightarrow^{\ell} M_2 \rrbracket = e_t : M_1 \Rightarrow^{\ell} M_2$$

$$M \sqcup M = M \qquad (M_1 \to M_2) \sqcup (M_3 \to M_4) = M_1 \sqcup M_3 \to M_2 \sqcup M_4 \qquad M_1 \sqcup M_2 = \star$$

$$d\langle M_1, M_2\rangle \sqcup d\langle M_3, M_4\rangle = d\langle M_1 \sqcup M_3, M_2 \sqcup M_4\rangle \qquad M \sqcup d\langle M_1, M_2\rangle = d\langle M \sqcup M_1, M \sqcup M_2\rangle$$

$$allFst\,(CS) = \{d.1 \mid d \in CS\} \qquad escDesc\,(G_1 \to G_2, \delta, CS) = allFst\,(CS) \qquad escDesc\,(G, \delta, CS) = \delta$$

Fig. 5. Cast insertion rules for the source language. We assume that pattern matches against the most specific case of $\sqcup$, and we omitted a dual to the last case of $\sqcup$.

5.2 Basic Translation Rules

The rule for Con is straightforward. For variable references, we need to distinguish between two cases: whether the referenced variable has only one version (the type is not variational in $\ddot{\Gamma}$) or two versions (the type for the variable in $\ddot{\Gamma}$ is a H variation). The first case, handled by Var-U, deals with variable references that have single versions only. The second case, handled by the rule Var-B, realizes the expanding phase mentioned earlier. Specifically, a variable reference is *expanded* into a variation between itself and its fast version. For example, applying this rule to the variable reference `adjust` in `adjust(f2)` yields a variation $H_a\langle$`adjust`, `adjust_Fast`$\rangle$.

We next turn our focus to translating function definitions. We again need to handle functions that have only single versions (the type for the function is not variational in $\ddot{\Gamma}$) and those that have both slow and fast versions (the type is an H variation), taken care by the rules Abs-U and Abs-B, respectively. In both rules, we use the relation $\vdash_M$ to translate the function body to make sure that the translated body contains no variations. We discuss $\vdash_M$ in Section 5.4. Essentially, it calls $\vdash_I$ to

translate the expression and then eliminates all variations. In Abs-B, we use the left alternative for generating its slow version and right alternative for the fast version. For example, given that $\Gamma(\texttt{adjust}) = H_a\langle \star \to \texttt{Int}, (\texttt{Int} \to \texttt{Int}) \to \texttt{Int}\rangle$, the slow version of `adjust` has the type $\star \to \texttt{Int}$, and its parameter has the type $\star$ when translating its body. The fast version, `adjust_Fast`, has the type $(\texttt{Int} \to \texttt{Int}) \to \texttt{Int}$, and thus the parameter `adjF` has the type $\texttt{Int} \to \texttt{Int}$ when translating its body.

The rule If for translating conditionals is mostly standard. We do not require branch types to be the same. Instead, the type of the conditional is the join ($\sqcup$) of its branch types (where the join of two different static types is $\star$). We do not use conditional types here as we did in Section 4.1 because using them has no advantages over using $\star$s, since a cast has to be always inserted whether a value has a conditional type or a $\star$. The rule uses conditionally inserted casts of the form $[\![e_t : M_1 \Rightarrow^{\ell_1} M_2]\!]$, which becomes e_t if M_1 and M_2 are the same, or a real cast on e_t otherwise.

5.3 Translating Function Applications

We now discuss the rule App for handling function calls. The auxiliary function $chcs(e_t)$ collects all variations in e_t. The superscript n on the second premise denotes the length of the sequence, allowing us to determine the number of the arguments. Other auxiliary functions are discussed in detail below. This rule essentially performs the refining phase discussed earlier, consisting of the following steps: (1) obtaining (likely variational) types for the function and arguments (premises 1 and 2), (2) splitting the type of the function into parameter types and a return type (premise 3), (3) computing whether the types of arguments and parameter types satisfy the subtyping relation (premise 4), (4) finding the best decision that goes to as many fast versions as possible for the function and the arguments (premise 5), which we refer to as the *best translation* (Note, a best translation does not necessarily have the best performance but it will use the most fast versions of functions in the translated expression), (5) handling function escape (premise 6), and (6) performing the selection (the $\lfloor\cdot\rfloor_{\delta_1}$ in the conclusion) to generate the translation result.

Splitting the function type (premise 3) The function $split(M, n)$ splits M into n parameter types and leaves the rest of M as the return type. This function is a combination of the `dom` and `codom` functions for decomposing gradual types from [Garcia and Cimini 2015]. In addition, this function handles variations, which causes some trickiness. For example, for the type $H\langle \star, \texttt{Int}\rangle$, we can split its first alternative into a function type but not the second alternative. The function *split* returns a pattern as the third component to indicate where splitting was successful (denoted by a $\top$) and where it failed (denoted by a $\perp$). For example, the pattern for splitting the type $H\langle \star, \texttt{Int}\rangle$ is $H\langle \top, \perp\rangle$. As an example of *split*, $split(\texttt{Int} \to \texttt{Bool} \to \texttt{Int} \to \texttt{Int}, 2)$ yields $((\texttt{Int}, \texttt{Bool}), \texttt{Int} \to \texttt{Int}, \top)$, where `Int` and `Bool` are the two parameter types. We defer the definition of *split* to Appendix D of the companion paper.

Computing subtyping results (premise 4) We define the operation $M_1 \lessdot M_2$ to determine if M_1 is a subtype of M_2 or not, which essentially implements the subtyping relation <: defined by Wadler and Findler [2009]. Since M_1 and M_2 are variational, we use a typing pattern π to collect more precise information rather than simply returning a binary result. The formal definition of $\lessdot$ is given below. When $\lessdot$ is called, we assume that the most specific case that matches arguments is used to perform the call. The left column specifies that base types U and $\star$ are subtypes of themselves; base types and $\star \to \star$ are subtypes of $\star$. Two function types satisfy the subtyping relation if their domains are contravariant "and" their codomains are covariant. Instead of using the logical "and" ($\wedge$) relation, we define an "and" relation $\otimes$ because the results from domains and codomains are

variational. The definition of $\otimes$ treats $\top$ as the logical truth value "true" and $\bot$ as the truth value "false" and extends naturally to handle variations.

$$\begin{aligned} U \lessdot U &= \top \\ \star \lessdot \star &= \top \\ U \lessdot \star &= \top \\ \star \to \star \lessdot \star &= \top \\ M_1 \to M_3 \lessdot M_2 \to M_4 &= M_2 \lessdot M_1 \otimes M_3 \lessdot M_4 \end{aligned} \qquad \begin{aligned} M_1 \to M_2 \lessdot \star &= M_1 \to M_2 \lessdot \star \to \star \\ d\langle M_1, M_3\rangle \lessdot d\langle M_2, M_4\rangle &= d\langle M_1 \lessdot M_2, M_3 \lessdot M_4\rangle \\ M_1 \lessdot d\langle M_2, M_4\rangle &= d\langle M_1 \lessdot M_2, M_1 \lessdot M_4\rangle \\ d\langle M_1, M_3\rangle \lessdot M_2 &= d\langle M_1 \lessdot M_2, M_3 \lessdot M_2\rangle \\ M_1 \lessdot M_2 &= \bot \end{aligned}$$

$$\top\otimes\pi = \pi \quad \bot\otimes\pi = \bot \quad d\langle\pi_1,\pi_3\rangle\otimes d\langle\pi_2,\pi_4\rangle = d\langle\pi_1\otimes\pi_2, \pi_3\otimes\pi_4\rangle \quad \pi_1\otimes d\langle\pi_2,\pi_4\rangle = d\langle\pi_1\otimes\pi_2, \pi_1\otimes\pi_4\rangle$$

The right column of the definition of $\lessdot$ deals with more complicated type forms. First, a function type whose domain or codomain is not a $\star$ is a subtype of $\star$ if the function type is a subtype of $\star \to \star$. For example, $\texttt{Int} \to \texttt{Int} \lessdot \star = \texttt{Int} \to \texttt{Int} \lessdot \star \to \star = \star \lessdot \texttt{Int} \otimes \texttt{Int} \lessdot \star = \bot \otimes \top = \bot$. Similarly, $\star \to \texttt{Int} \lessdot \star = \star \to \texttt{Int} \lessdot \star \to \star = \star \lessdot \star \otimes \texttt{Int} \lessdot \star = \top \otimes \top = \top$. The next case deals with two variational types that have the same outermost variation, and the definition applies to their corresponding alternatives. When only one type is variational or they are both variational but having different outermost variations, they are made to be variational with the same outermost type by exploiting the fact that M_1 is equivalent to $d\langle M_1, M_1\rangle$. For example, $\texttt{Int} \lessdot d\langle\texttt{Bool}, \star\rangle = d\langle\texttt{Int} \lessdot \texttt{Bool}, \texttt{Int} \lessdot \star\rangle = d\langle\bot, \top\rangle$. If none of the above cases applies, the result of $\lessdot$ is $\bot$.

Determining the best decision (premise 5) Having obtained the result about subtyping and stored it in π, the goal here is to find a decision δ from π such that selecting the function application with δ yields the best translation. Before we start, we assume that the outermost variation in π is the same as the variation for the function that is being called. This is achievable because π is equivalent to $d\langle\lfloor\pi\rfloor_{d.1}, \lfloor\pi\rfloor_{d.2}\rangle$ for any variation d. We use O to refer to the outermost variation.

To find the best decision, we first make an important observation between a decision and the resulting translation. Specifically, assume `f` appears in the expression before translation, d is the variation introduced for `f`, and the translation output e_t is obtained by selecting the expanded variational expression with the decision δ. Under this assumption, `f` appears in e_t if $d.1 \in \delta$, and `f_Fast` appears in e_t if $d.2 \in \delta$. Based on this observation, the decision for the best translation should contain as many right selectors ($d.2$s) as possible. Consequently, it seems that a decision δ for π is best if selecting π with δ leads to the rightmost $\top$ in π. We use the notation δ_r to refer to this decision. For example, for the pattern $O\langle D\langle B\langle\top, \bot\rangle, \bot\rangle, D\langle\bot, B\langle\top, \bot\rangle\rangle\rangle$, the δ_r is $\{O.2, D.2, B.1\}$.

Surprisingly, this strategy indeed finds the best decision. We express this idea through Theorem B.2 and give its proof in Appendix B.1 of the companion paper (Note that the Theorem holds only for typing patterns generated from $\lessdot$. It holds because for each variational type, the left alternative is always less precise than the right alternative). We next define a function *rgtMost* for computing the δ_r for π as follows. In the definition, $\varnothing_\bot$ and $\varnothing_\top$ denote empty sets but with different tags.

$$rgtMost\,(\bot) = \varnothing_\bot \qquad rgtMost\,(\top) = \varnothing_\top$$

$$rgtMost\,(d\langle\pi_1, \pi_2\rangle) = \begin{cases} \{d.2\} \cup rgtMost\,(\pi_2) & rgtMost\,(\pi_2) \neq \varnothing_\bot \\ \{d.1\} \cup rgtMost\,(\pi_1) & rgtMost\,(\pi_1) \neq \varnothing_\bot \\ \varnothing_\bot & \text{otherwise} \end{cases}$$

For a variational pattern, we extend the rightmost decision from the right subtree (if it is not $\varnothing_\bot$) with $d.2$ and return the extended decision. Otherwise, we check the rightmost decision from the left subtree, extend it with $d.1$, and return it. If neither subtree returns a rightmost decision, we return $\varnothing_\bot$.

However, in addition to the rightmost decision, the best decision needs to consider two cases. In the first case, π is a $\bot$, which means that the parameter type and the argument do not satisfy

```
50  def escape(f):
51    adjust(f)
52    return f
53
54  escape(f1)(true)
```

```
55  def f3(x):
56    x + 2
57    return f2
58
59  f3(6)(7)
```

```
60  def f4(x):
61    x + 2
62    return x
63
64  f2(f4(5))
```

Fig. 6. Three programs that exhibit trickiness on function escape (left), remaining variations (middle), and performance slow down (right). The middle and right programs will be used in Sections 5.4 and 6.2, respectively.

the subtyping relation. One such example is `adjust(notB)`. Here the parameter type of `adjust` is $A\langle\star, \texttt{Int} \rightarrow \texttt{Int}\rangle$, the type of `notB` is $\texttt{Bool} \rightarrow \texttt{Bool}$, and $\texttt{Bool} \rightarrow \texttt{Bool} \lessdot A\langle\star, \texttt{Int} \rightarrow \texttt{Int}\rangle$ is $A\langle\bot, \bot\rangle$. In this case, the rightmost decision is $\varnothing_\bot$ but we still need to eliminate the variations in the application. We should eliminate the variations by taking their left alternatives, which does not optimize the expression.

In the second case, a variation that appears in a function application may not appear in the rightmost decision for the corresponding pattern. For example, for the expression `adjust(f1)`, the pattern is $H_a\langle H_{f1}\langle\top, \bot\rangle, \top\rangle$. The rightmost decision for this pattern is $\{H_a.2\}$, which does not include the variation H_{f1}. For all such variations, we can include their right selectors in the best decision. The reason is that for any π, if $\lfloor\pi\rfloor_\delta = \top$, then $\lfloor\pi\rfloor_{\delta_1} = \top$ for any $\delta \subseteq \delta_1$.

Considering the above two cases, we define the function *bestDesc* to compute the best decision as follows.

$$bestDesc\,(\pi, CS) = \begin{cases} allFst\,(CS) & rgtMost\,(\pi) = \varnothing_\bot \\ rgtMost\,(\pi) \cup \{d.2 \mid d \in CS \wedge d.1 \notin rgtMost\,(\pi)\} & \text{otherwise} \end{cases}$$

where CS should contain the set of all variations in the expression. The function *allFst* is defined at the bottom of Figure 5.

Handling function escape (premise 6) If selecting the return type of the function (M_3) with the best decision is not a function type, then we can already generate the target expression. As a result, this premise has no effect, as we directly assign δ to δ_1. However, some trickiness can rise if selecting M_3 with the best decision yields a function type.

To illustrate, consider the program in Figure 6 left. Based on the definition of `escape`, we can calculate that its slow version has the type $\star \rightarrow \star$ and the fast version has the type $(\texttt{Int} \rightarrow \texttt{Int}) \rightarrow \texttt{Int} \rightarrow \texttt{Int}$. When calling `escape` with `f1` (line 54), we can translate this call to either `escape(f1)` or `escape_Fast(f1_Fast)`, based on their types. The best decision will yield the translation `escape_Fast(f1_Fast)`. During execution, this call will return `f1_Fast`. When calling `f1_Fast` with `true` (line 54), a runtime error will be encountered on line 2 (Figure 1). In contrast, the desired behavior is that a cast error should be detected on line 6 for the code `escape(f1)(true)`.

The reason for this unexpected behavior is that the decision for translating `escape(f1)` to `escape_Fast(f1_Fast)` is premature as the returned value is a function, which may be applied to some argument later that does not satisfy the requirement for the translation to go to fast paths. Our solution here is that we make a conservative decision for such cases. Specifically, if the result of an application has a function type, we conservatively use the slow version of the functions, as if we do not optimize the function call. Following this idea, we use slow versions of both `escape` and `f1` for translating `escape(f1)`.

We implement this idea in the premise 6 of the rule App through the function *escDesc*, defined at the bottom of Figure 5. The parameter G is the result type of the application, δ is the best decision from the previous premise, and CS is a set of variation names.

Generating the best translation The App rule then generates the target expression by selecting with the best decision δ_1.

5.4 Eliminating Remaining Variations and Properties

The target expressions generated by the translation relation $\ddot{\Gamma} \vdash_I e \rightsquigarrow e_t : M$ may still contain variations. To illustrate, consider the program in Figure 6 middle.

The function `f3` will be translated into two versions since its parameter is optimizable with the type `Int`. During translation, the reference to `f2` will be translated to $H_{f2}\langle$`f2`, `f2_Fast`$\rangle$. We might wonder why the variation H_{f2} is not eliminated even after the whole function has been translated. The reason is that App is the only rule that eliminates variations and `f2` is not applied.

We obviously need to eliminate this choice, but which alternative of H_{f2} should we use to eliminate it? If we use the second alternative, then `f2` is optimized in the body of `f3`, which violates soundness. To illustrate, consider `f3` was called on line 59. Returning the function `f2_Fast` from `f3(6)` means that inside `f2` no cast will be inserted to protect the call to `notB` and the `Int` value `7` will be incorrectly passed to `notB` (which expects Boolean arguments).

As a result, we should always use the first alternatives for remaining variations. We express this idea in the rule Main through the *allFst* function.

We now investigate the properties of our translation. First, an expression that passes static gradual type checking always generates a target expression, as expressed in the following theorem.

Theorem 5.1. *For any e, if $\pi; \Gamma \vdash e : M \mid \Omega$ and $\lfloor \pi \rfloor_{allFst(\pi)} = \top$, then $\ddot{\Gamma} \vdash_M e \rightsquigarrow e_t : G$ for some e_t.*

Next, our translation preserves program behaviors, as expressed in the following theorem, where $\vdash_S$ denotes a standard cast insertion process [Siek et al. 2015b], $\Downarrow$ denotes a standard evaluation semantics for cast languages [Siek et al. 2015b], and $\Uparrow$ denotes that the evaluation is divergent. Also, we use the function *unbox* to discard casts associated with values [Siek and Taha 2006]. The definition of v is standard [Siek et al. 2015b] and we omit it here.

Theorem 5.2 (Correctness of optimization). *Given a closed expression e, $\ddot{\Gamma} \vdash_M e \rightsquigarrow e_{1t} : G_1$, and $\varnothing \vdash_S e \rightsquigarrow e_{2t} : G_2$, then $e_{2t} \Downarrow v_2$ implies $e_{1t} \Downarrow v_1$ and $unbox(v_2) = unbox(v_1)$, $e_{2t} \Downarrow$ `blame` ℓ implies $e_{1t} \Downarrow$ `blame` ℓ, and $e_{2t} \Uparrow$ implies $e_{1t} \Uparrow$.*

Intuitively, our optimization is correct because it only eliminates casts that satisfy subtyping relations, and based on blame-subtyping theorem [Wadler and Findler 2009] such casts cannot give rise to runtime failures. The proof can be constructed by induction over standard cast insertion rules [Siek et al. 2015b] and those from this paper (Figure 5). We provide the proof in Appendix B.2.

6 IMPLEMENTATION AND EVALUATION FOR PYTHON

In this section, we discuss our implementation of DT on Reticulated Python [Vitousek et al. 2014] and use this implementation for evaluating to which degree DT improves performance.

6.1 Implementation, Benchmarks, and Experimental Setup

We implemented DT by extending the Reticulated type checker and guarded cast insertion processes, adapting our formalization to handle additional Python features like objects, classes, and local variable inference. In our implementation, we represent objects and classes as mappings from names, including fields and methods, to types. We infer types for class methods as for functions but extend them with a parameter whose type is the class name. Hereafter, we refer to Reticulated with and without DT as DT and Reticulated, respectively. To compare the performance of DT with IAO [Rastogi et al. 2012], we have implemented a version of it in Reticulated.

	chaos	even/odd	fft	float	MC	nbody	pascal	pi	ray	SOR	sieve	spectral
LOC	319	29	133	64	91	157	70	68	448	110	56	85
Retic	0.23	0.01	0.06	0.02	0.03	0.04	0.03	0.02	0.09	0.02	0.03	0.09
DT	0.71	0.04	0.26	0.12	0.15	0.26	0.20	0.09	0.58	0.15	0.22	0.11
Inferred	45	100	57	50	35	38	68	80	71	33	46	50

Fig. 7. Program size (second row) of the evaluated benchmarks, translation times (in seconds) of Reticulated (third row) and DT (fourth row) for them, and percentages (%) of types inferred for the untyped program.

Details about the benchmarks we use to evaluate DT are presented in Figure 7. The programs are mainly adapted from the Python benchmark suite (these programs were also used by Vitousek et al. [2017], Campora et al. [2018a], and Greenman and Migeed [2018]). The three programs that are not from that suite are even/odd, which illustrated the space efficiency problem of proxies in Herman et al. [2010], the sieve program from Takikawa et al. [2016], and the pascal program that was adapted from Rosetta Code's "Pascal Matrix Generation" and "Permutations" entries.

To give an idea about the overhead of the cast-insertion process in DT (not runtime duration of the cast inserted programs), we provide the timing data in Figure 7. We use the dynamic configurations for timing the overhead, since the lack of type annotations in the program yields more work for the type inference algorithm than other configurations. In our initial implementation, we observed that the translation of DT often took about 4-12 times more than using Reticulated. We then profiled the type inference and the translation processes to see if we can improve that result. The profiling revealed that a major portion of the time was spent on variational unification, for finding optimizable parameters and inferring their types. The main reason this happens is that we did not simplify variational types often enough, making variational types to build up quickly and the computations with them inefficient. We have added simplification code (for example reducing $A\langle \texttt{Int}, A\langle \texttt{Char}, \texttt{Bool}\rangle\rangle$ to $A\langle \texttt{Int}, \texttt{Bool}\rangle$ since the alternative `Char` can never be reached [Chen et al. 2014]) to our implementation. We observed the the translation overhead dropped to 3.1-7.3 times of the Reticulated's translation time.

To further test the scalability of DT, we measured the times for 50 programs with their LOCs range from 500 to 10K, and the overheads for them over Reticulated are consistently within 4.5 to 7.1 times. For example, for the program that is 5k, the time for DT translation is 48.3 seconds while that for Reticulated is 8.5 seconds. For the program that is 10k, these numbers are 104.6 and 18.1 seconds, respectively. Since the overhead is comparable to the that of recent work on employing variational analyses for gradual typing [Campora et al. 2018b,a], we do not explore this issue further.

In the following subsections, we evaluate the effectiveness of DT by comparing its performance with Reticulated and IAO. For small programs (those having less than 2^{10} total configurations) we evaluate performance by executing and timing all of their configurations (Here each configuration represents a unique way of whether assigning types to the parameters). For large programs, we randomly sample 2^{10} configurations to measure, since the work by Greenman et al. [2019] indicates that this sufficiently represents a more global view of performance. All results were collected on a server with virtualized 64 bit Ubuntu 16.04 on a 2.6 GHz processor with 1 core and 2 GB of RAM. We use Python 3.5. Each time is an average of 10 runs.

6.2 DT Versus Standard Reticulated and IAO Across All Configurations

We first compare the performance of DT with that of Reticulated and IAO across all 6352 configurations in 12 benchmarks. For any program P, if Reticulated translates it to P_R and DT translates

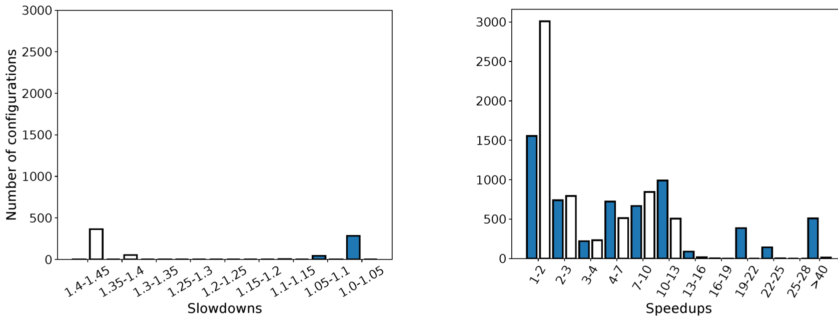

Fig. 8. Slowdowns (left) and speedups (right) for DT compared to Reticulated (filled bars) and IAO (unfilled bars) across all configurations. The ratio values appear between the two consecutive bars.

it to P_D, then the speedup over Reticulated is the running time of P_R divided by that of P_D. The slowdown is computed conversely. The speedups and slowdowns over IAO are calculated similarly.

These comparisons give us a perspective on how well DT works compared to previous approaches. The distribution of these slowdown and speedup results are depicted in Figure 8. From the figure, we observe that DT typically achieves much better performance when compared to Reticulated. Specifically, DT enjoys more than 40 times speedups in about 8% of configurations, 4 - 25 times speedups in 48% of configurations, 1 - 4 times speedups in 39% of configurations. We further investigated configurations that DT has a speedup of 1 - 2 times. The reason is that, even though we ran each configuration 10 times and averaged the running time, speedups and slowdowns maybe because minor time variations. For example, for one configuration in the monte carlo benchmark the time of DT is 0.164s while that of Reticulated is 0.169s. We measure how often speedups are insignificant. Among all configurations that DT has a speedup between 1 - 2, 348 have a speedup between 1.01-1.1, and more than 1200 configurations have a speedup of more than 1.1. This indicates that for such speedups, the speedups are mostly due to performance optimization, rather than due to time variations. Finally, DT suffers slowdowns in only 5% of configurations, and the slowdowns are all minor, within 1.15.

Compared to IAO, DT again performs much better. Specifically, DT experiences 4 - 25 times of speedups in 30% of configurations and 1 - 4 times of speedups in 63% of configurations. It suffers slowdowns in only 7% of configurations, and the largest slowdown is 1.45 only. Overall, these results show that DT significantly improves performance over Reticulated and IAO.

To illustrate why DT can actually slowdown performance, consider the function `f4` in Figure 6 right. Without optimization, no cast will be inserted when passing the result from `f4(5)` to `f2` since both types are $\star$. However, in DT, the fast version of `f4` will be used to perform the call, which will return an `Int` value. When that value is passed to `f2`, a cast from `Int` to $\star$ will be inserted. Optimization may slowdown performance, but in very few cases, as our evaluation has shown.

6.3 Overhead Compared to Untyped Configurations

This section studies the performance of DT with a focus on individual benchmarks. One important question is, how well does DT normalize the overhead of mixed type configurations toward the dynamically typed configuration? This question is important since Takikawa et al. [2016] noted that the mixed type configurations of gradual programs can often have unacceptable levels of overhead, sometimes more than 100 times. Therefore, for each benchmark we time the execution of Reticulated, IAO, and DT across all configurations and compare them to the time taken to execute DT's dynamic configuration. The results of this evaluation are presented in Figure 9.

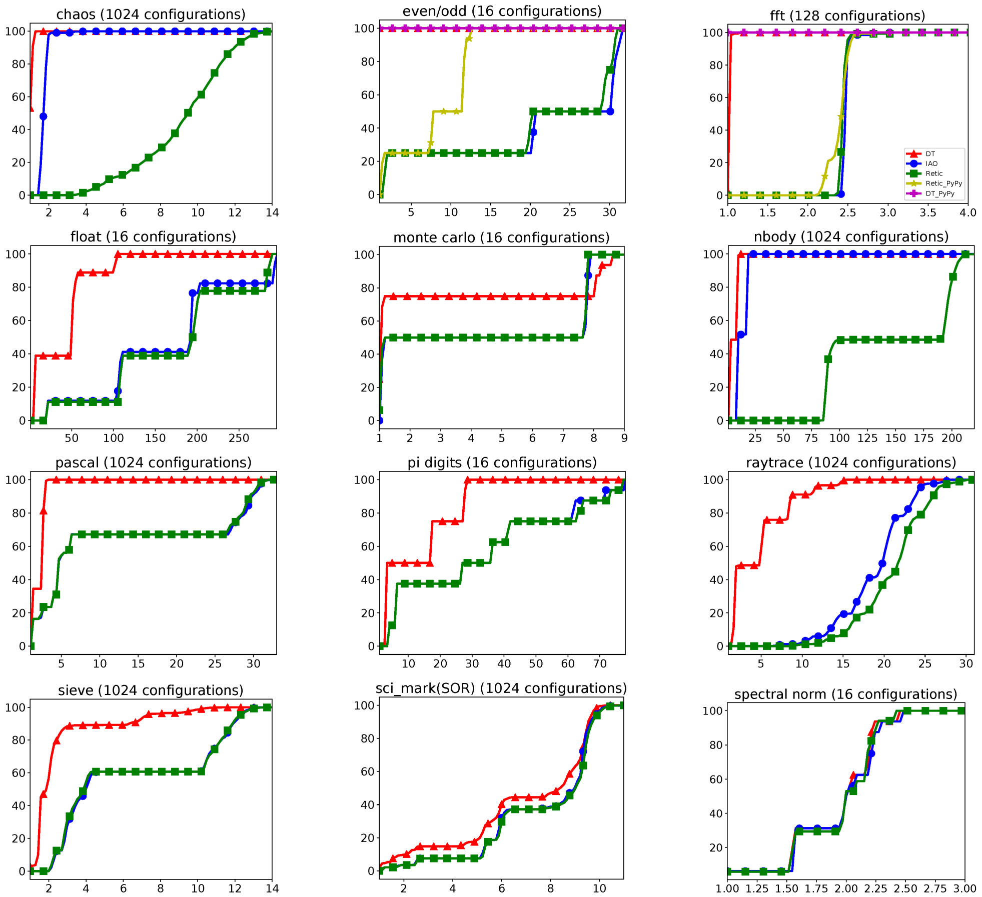

Fig. 9. The slowdown of different configurations when running Reticulated, DT, or IAO when compared to running DT on the dynamic configuration. The x-axis measures slowdown, while the y-axis measures the percentage of configurations that have less than this slowdown. The line and mark for Reticulated is green with squares, for DT is red with triangles, for IAO is blue with circles, for PyPy + Reticulated is yellow with stars, and for PyPy + DT is magenta with crosses. Note the red and magenta lines almost completely overlap. Only even/odd and fft have results with PyPy since PyPy encounters runtime errors on all other programs. A legend is given in fft plot and is consistent across all plots. The baseline of this figure is running Reticulated on the untyped configuration, where Reticulated incurs the least overhead.

From Figure 9, we make several important observations. First, we observe that DT generally outperforms Reticulated across the 12 benchmarks, and DT performs markedly better on chaos, even/odd, fft, float, nbody, pascal, pi digits, and raytrace. For all benchmarks except chaos and nbody, IAO's performance falls more in line with Reticulated than with DT. Compared to IAO, DT performs much better for most benchmarks except chaos, sci_mark, and spectral norm. Note that although the curves of DT and IAO look close for nbody, IAO has significant slowdowns of over 10 times when compared to DT.

In the chaos benchmark, the performance of DT and IAO are close because functions and methods there seldom interact, and few optimization opportunities exist for DT. Most other benchmarks contain more interacting functions where performance is greatly improved by calling fast versions of functions in DT. In practice, we expect that functions and methods interact very often.

Second, we observe that DT brings many more configurations into having less than 3 times slowdown than Reticulated and IAO (these configurations are called 3-deliverable using the terminology from Takikawa et al. [2016]). For pi digits and raytrace, DT has many more configurations between 3 to 10 times slowdown (such configurations are called 3/10-usable). In float, nbody, and raytrace we see that many configurations are 3-deliverable where none were in Reticulated or IAO. In particular none of Reticulated's or IAO's configurations were even 3/10-usable for float. Though many IAO configurations were 3/10-usable for nbody, almost twice as many were either 3-deliverable or 3/10-usable from DT.

Third, we observe that DT rarely performs worse than IAO and Reticulated. One of the few examples is the monte carlo benchmark, and we took a close look into this benchmark. Among 16 configurations, DT has a speedup of about 9 times for 4 configurations and has a speedup of 0.96 to 1.05 for 11 configurations. For the latter 11 configurations, the runtimes are very close between DT and Reticulated. For example, for one configuration, the runtime of DT is 0.175s while that of Reticulated is 0.174s. We further looked at the translated programs and observed that they are the same across all 11 configurations. For the last configuration, the speedup is 0.91. We looked at the translated programs and observed that in one place, DT's translation incurred slowdown (from 1.31s to 1.44s), similar to the scenario in Figure 6 right.

All approaches performed nearly identically on the spectral norm benchmark, even though DT is able to infer about 50% of types. To understand this phenomenon, we looked through many translated programs and realized that the main reason is that values produced from libraries or built-in functions are untyped. When these values are used as arguments for function calls, it is impossible to move to fast path calls even though we generated fast versions. For example, each call to `eval_A` in the program has an argument produced by the built-in function `enumerate` and so `eval_A_fast` is never called.

Finally, DT typically has a large number of configurations close to the y-axis than other approaches have. This means that DT is more successful at normalizing performance to the dynamic configuration than other approaches, though for some benchmarks and configurations it still suffers from much overhead.

Optimality of Optimization The results in Figure 9 demonstrates that DT significantly improves gradual typing performance. On the other hand, Figure 8 shows that DT also slows down programs in some cases. As such, it is interesting to know if the optimality of optimization, which means that the optimization never slows down programs, could ever be achieved.

To answer this question, we performed an experimental study. Specifically, we randomly sampled 300 configurations for which DT slowed down performance. We observed that, among them, for 278 configurations DT outputs are the same as normal cast insertions, and the slowdown are due to small time variations across different runs. Even though we ran 10 times for each configuration and averaged the running time, such small time variations are hard to totally eliminate (see the discussion below Figure 8 for why our speedup result is still valid in presence of such time variations). For the other 22 configurations, DT inserted extra casts but they are all of the form $e : U \Rightarrow \star$ for some expression e. One such example was given in Figure 6 (right) where the fast path inserts a cast $\texttt{f2(7} : \texttt{Int} \Rightarrow \star\texttt{)}$ that is absent from the normal path. It is thus tempting to eliminate all such casts in the cast inserted program since casts like $7 : \texttt{Int} \Rightarrow \star$ should never fail.

However, this idea does not generalize. To illustrate, consider a higher-order cast $e : \texttt{[Int]} \Rightarrow \star$. Such casts could happen when the return type from some fast version is `[Int]` and the return value is passed to some slow version function whose parameter type is $\star$. The cast $e : \texttt{[Int]} \Rightarrow \star$ should not be eliminated because casts involving the list type will install a proxy which makes sure that all the elements read from the list and added to the list will have the type `Int`. While we may be

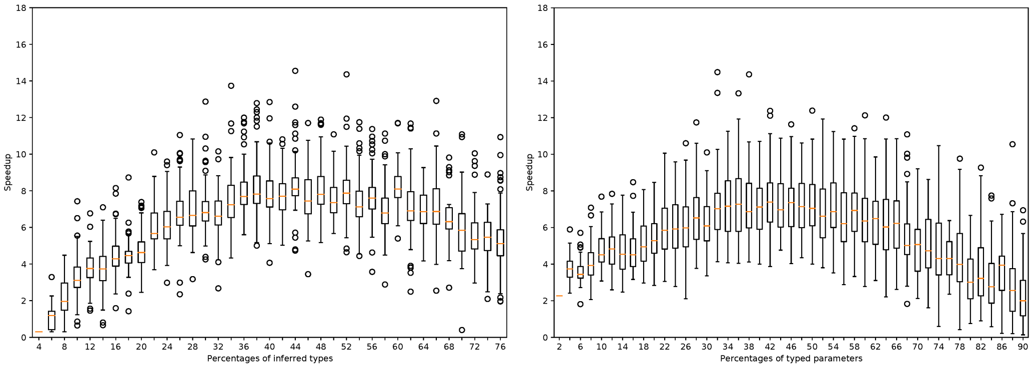

Fig. 10. Relations between speedups and inferred types (left) and (annotated types) for DT compared to Reticulated for the raytrace benchmark. While our approach infers only 71% of types for the untyped configuration, it can infer higher percentages for some configurations with certain type annotations present.

able to statically infer that the list currently contains only `Int` values, we can not guarantee that elements added to the list also have the type `Int`, particularly when the list is passed to a function whose parameter type is a ⋆. As such, the proxy could not be removed and the cast e : [`Int`] ⇒ ⋆ should not be eliminated. In general, this is the case for all higher-order casts.

In general, it is desirable that an optimization never slows down programs. In practice, however, this is very hard to achieve. For example, existing gradual typing optimization approaches [Feltey et al. 2018b; Moy et al. 2021; Ortin et al. 2019] all slow down certain programs.

6.4 Types and Performance

We next explore the correlations between inferred types and speedups of DT. We present the results for the raytrace benchmark in Figure 10. The results of this benchmark are typical across other benchmarks except for sci_mark and spectral norm. In Figure 10 (left), the x-axis represents the percentage of parameters whose types can be inferred over all parameters. Figure 10 (right), the x-axis represents the percentage of parameters with type annotations over all parameters.

From Figure 10 (left), we observe two interesting phenomena. First, for a given percentage of inferred types, the speedups vary significantly. There are several possible reasons for this. (1) While two configurations may have the same percentage of inferred parameters, the parameters where types are inferred can be very different, based on what annotations are already present in the configurations. For example, for one configuration parameters of repeatedly called functions are inferred while for another configuration parameters of rarely called function may be inferred. (2) Configurations with similar inferred type ratio may have very different runtimes from Reticulated. Since speedup is calculated as the runtime of reticulated over that of DT, differences in Reticulated runtimes lead to very different speedups. (3) The program structure, including the library and built-in functions used, can significantly impact the runtime of configurations. For example, if arguments to functions whose parameter types are inferred have the dynamic types, then the inferred parameter types do not help with the performance because the fast version is not used due to dynamic argument types.

Another phenomenon is that the speedups first increase as more types are inferred and then decrease. There are a few reasons for that. (1) When few types are inferred, we further need to consider two cases. In the first case, most parameters already have type annotations, which means that the configuration already has good performance, and there are not many opportunities for improvement. In the second case, the type inference fails to infer more types, and consequently

the performance improvement is not very significant. (2) When relatively more (25%-65%) types are inferred, there are more opportunities to go to the fast versions of the functions. Moreover, these configurations likely have many type annotations present and Reticulated is likely to have most significant slowdowns due to many inserted casts. (3) When more types can be inferred, the original configurations have relatively few type annotations. Reticulated does not have too much overhead for such configurations and there are not many opportunities for improvement.

From Figure 10 (right) exhibits a similar pattern as From Figure 10 (left) does. The main reason is that the amount of inferred types and that of type annotations are complement to each other. As a result, the reasons for Figure 10 (right) are similar and we will not present them in detail.

6.5 Further Evaluations and Discussions

Function proxies Guarded Reticulated installs a proxy for each function to respect its annotation when it is called by external code. We are able to remove these proxies in our fast paths for functions without sacrificing any safety guarantees since external code always call into the slow versions of functions. An interesting related question is, does DT eliminate further casts in addition to the removed proxies? To answer this question, we removed function proxies in all three implementations and counted the number of casts executed across all configurations. We plot the result and present the plots in Appendix A.1 of the companion paper to save space. Our main observation is that the curve trends in the plots match those in Figure 9. Moreover, in certain benchmarks the distances among curves stay similar and in others they become closer. This indicates that DT does eliminate additional casts and that these eliminations sometimes provide significant performance gains. For example, compared with IAO for the float benchmark, the ratios (number of casts in IAO over that in DT) are more than 20 times in 25% of configurations and more than 10 times in 81.25% of configurations. Since in most cases the set of casts in DT is a subset of those in Reticulated and IAO, it indicates DT spends less time on executing casts.

Memory consumption We have measured memory consumption for all the benchmarks. We present the results in Appendix A.2 of the companion paper. Compared with IAO translations, DT translations always consume less memory. Compared with Reticulated, DT translations consume more memory in only two benchmarks and is within a factor of 1.5 and consume 25% to 100% of memory of Reticulated translations do in all other cases.

Relation with JIT We have tested PyPy (a Python JIT) + Reticulated and PyPy + DT on benchmarks, and we plotted the results in Figure 9. We were only able to successfully run PyPy on even/odd and fft since PyPy encountered errors on all other cast-inserted programs. The error happens when casts from $\star$s to objects are encountered in PyPy. Our main insights from Figure 9 are: (1) DT outperforms PyPy on optimizing gradual programs, as can be seen from the results of DT and PyPy + Reticulated, and (2) PyPy helps further optimize the performance of DT on some configurations, as lines DT and PyPy+DT show. Overall, DT and PyPy seem to be complementary.

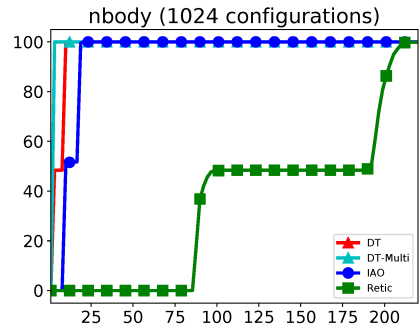

Fig. 11. The performance comparison between DT, DT-Multi, IAO, and Reticulated.

Multiple fast versions The approach and evaluation so far generate only one fast version for each function. This may miss some optimization opportunity if a function takes multiple arguments and only some argument type satisfies the fast version requirement and others do not. One way

to remedy this issue is to generate multiple fast versions for a single function. Specifically, for a function with n parameters, we may generate up to $2^n - 1$ fast versions. While generating multiple fast versions can potentially improve the performance, it also increases the code size, no more bound by a factor of 2. To investigate the tradeoff between performance gain and code size increase, we have explored the impact of generating multiple fast versions.

Adapting our approach and prototype to generate multiple fast versions is surprisingly simple. Specifically, instead of creating a variation with two alternatives for types of the slow and fast versions of each function (Section 4.3), we create a different variation for types of each parameter of the function. This allows us to create a slow and fast version for each parameter. We name DT that generates multiple fast versions as DT-Multi.

We have compared the performance of DT-Multi, DT, Reticulated, and IAO and showed the result for nbody in Figure 11. We can observe that DT-Multi further normalizes the performance of Reticulated towards the dynamic configuration, indicating that there are indeed some benefits of generating multiple fast versions from the performance perspective. As for code size, DT-Multi increases it by a factor of 16.7 if we eagerly generate all fast versions no matter whether they are used or not. If we instead generate a fast version only when it is called with corresponding argument types, then the code size increases by a factor of 2.7. This means that there are not too many different type combinations of calling fast versions.

We have also investigated other benchmarks. While they showed a similar pattern with regard to code size increase, they did not show much performance improvement. This can be attributed to a few reasons. First, not all parameters can be inferred to be optimizable, limiting the number of fast versions that could be created. Second, performance is often determined by higher-order parameters (parameters that accept function types, lists, etc.). In some cases, a function has none or only one such parameter. The performance will not be affected by the number of fast versions generated. Finally, if static types are inferred for none of the argument or all of the arguments, then making multiple fast versions will not improve the performance.

In general, generating multiple versions can enjoy more performance gains, the degree of which, however, can be affected by many factors.

7 IMPLEMENTATION AND EVALUATION FOR GRIFT

Kuhlenschmidt et al. [2019] recently proposed Grift and implemented it from scratch rather than translating it to an underlying dynamic language. Grift programs are compiled to C programs to eliminate possible interference of dynamic language features on gradual typing performance. In addition, it uses coercions [Henglein 1994] to improve both time and space efficiency of gradual typing. For example, Kuhlenschmidt et al. [2019] reported that coercions based gradual typing implementation eliminates catastrophic slowdowns experienced by typed-based implementations.

Overall, Grift is regarded as a close-to-the-metal implementation of gradual typing. We are interested in knowing if DT can be combined with Grift to further improve its performance. To facilitate such an investigation, we have implemented our ideas in this paper into Grift. In this section, we use DT and Grift to refer to the implementations of Grift with and without our extension, respectively. The times in this section are measured on a laptop with 64 bit Ubuntu 20.04 on a 2.6 GHz processor and 32 GB of RAM. We used Racket version 7.9. Each time is an average of 10 runs.

We have considered 9 benchmarks for performance evaluation, including all 8 used in Kuhlenschmidt et al. [2019] for evaluating their implementation plus the CPS-style even/odd program from that paper. We defer details of these benchmarks to Kuhlenschmidt et al. [2019]. For each benchmark, we evaluate the same configurations that are used for evaluation in Kuhlenschmidt et al. [2019]. Overall, our evaluations involved 7948 configurations. Figure 12 gives the benchmark names and the number of configurations we evaluated for each benchmark.

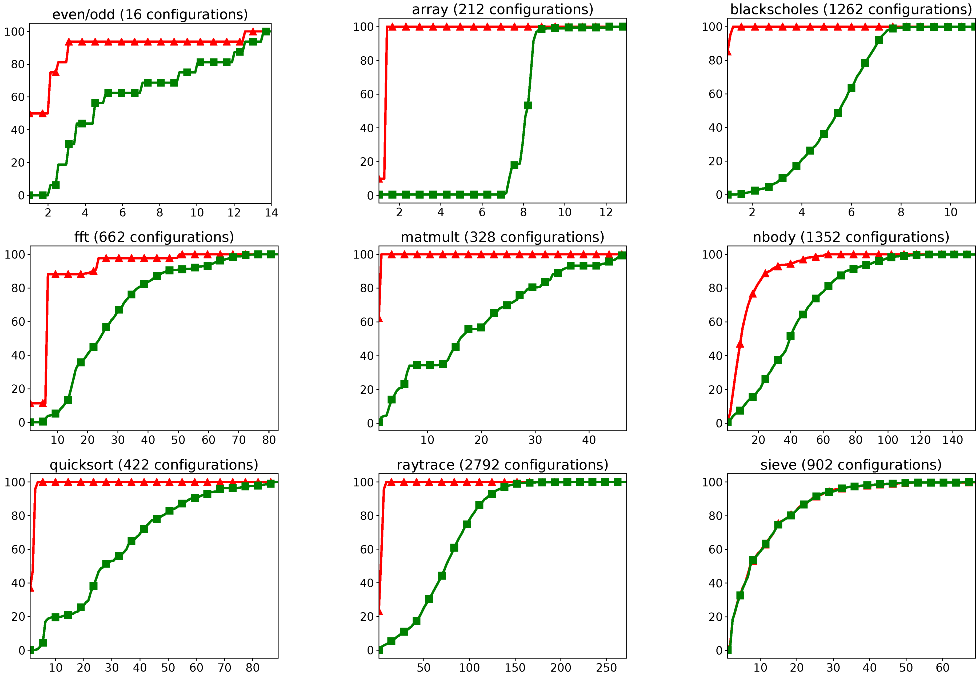

Fig. 12. The performance of Grift and DT on partially typed configurations. A point (x,y) on a curve means that about y% of configurations have less than x times slowdown compared to the fully static configuration. The line and mark for Grift is green with squares and for DT is red with triangles.

Figure 12 presents the performance comparison between DT and Grift for individual benchmarks. In these figures, we use the fully static configuration as the baseline for measuring slowdown because that configuration is always the quickest among all configurations. In contrast, the fully dynamic configuration is almost the slowest for several benchmarks, presenting results using the dynamic configuration as baseline will be very uninformative.

In general, DT performs much better for Grift than for Python. The main reason is that DT is able to infer more types for Grift benchmarks, due to the following reasons.

(1) The Grift language introduces many different operations for different types. For example, it uses `+` for addition of `Int` values and `fl+` to add `Float` values. This reduces ambiguity and allows us to infer more precise parameter and variable types.
(2) For all `if` expressions in these benchmarks, either variables are used in only one branch or they have the same type when they are used in different branches. This allows us to infer fully static types for them.
(3) Many functions are parametric, such as `swap` that swaps two elements in a given vector in the `quicksort` benchmark. When looking at `swap`, while we can infer the types for the parameters that serve as indexes to be `Int`, we can not infer the element type of the vector, so we assign the parameter type to be `Vect a`. At the use site, if we determine that the type of the argument to `swap` is `Vect Int`, we create a fast version of `swap` for `Vect Int` and use the fast version at the call site. This enables to infer more types.

From Figure 12, we observe the following. First, DT eliminates almost all slowdown in the even/odd, array, blacksholes, fft, matmult, quicksort, and raytrace benchmarks, making almost all configurations perform similar to the fully static configuration. Second, DT brings a large portion

of configurations close to the fully static configuration in nbody. Finally, DT performs almost identically to Grift on sieve. The reason is that sieve uses equirecursive types to annotate streams, and our current type inference does not infer such types. As a result, DT could not help improve Grift. Overall, we believe that DT is viable for improving the performance of Grift and helping bring the performance of gradual typing closer to static typing.

8 RELATED WORK

Performance improvement through type inference or static analysis In the area of dynamically typed languages, type inference has been employed to improve performance by utilizing type-based optimizations [Cartwright and Fagan 1991; Henglein 1994]. In this work, we target gradually-typed programs. IAO [Rastogi et al. 2012] was the first approach using type inference to optimize gradually-typed languages. The work on gradual cost analysis [Campora et al. 2018a] used inference and cost analysis to recommend performant configurations to programmers. To realize the performance gains, a programmer must annotate programs as recommended, while our approach automatically optimizes performance. Moreover, the recommended type annotations may change program behaviors while DT always preserves program behaviors.

Static analysis has been employed to optimize the performance of cast-inserted programs by removing installed casts that always succeed. Such work includes modular set-based analysis [Meunier et al. 2006] and soft contract verification (SCV) [Nguyen et al. 2017, 2014]. Unlike IAO and our approach, these approaches optimize programs without considering open-world soundness. As a result when the optimized functions are called by existing code, their behaviors may not be preserved. Finally, the work on optimizing transient gradual typing [Vitousek et al. 2019] uses inference on the cast-inserted program to remove casts verified by inference. Though their approach supports open-world soundness, their evaluation did not enforce open-world soundness. Since that work followed the spirit of IAO, the shortcomings we identified in Section 1 apply to it. It is thus unclear how well that approach or SCV optimizes programs in an open-world setting. In contrast, DT can handle the open world by keeping slow versions of functions, that when proxied preserve soundness (sans object and type identity issues [Vitousek et al. 2014]).

Performance improvement through language design Other optimization approaches work by either designing source languages with certain properties or by changing the soundness goals of the cast language. An example of the former approach is the Nom programming language where all values of the source language inherently have a static type upon construction [Muehlboeck and Tate 2017] (similar approaches have been designed for TypeScript [Rastogi et al. 2015; Richards et al. 2015]). This means that the proxies in Racket and guarded Reticulated are unnecessary in Nom. An example of the latter approach is transient gradual typing, which eliminate proxies by performing only first-order checks that provide weaker safety guarantees. We conjecture that our idea of generating multiple versions of a function can be adapted to transient, though we focused on optimizing the more common guarded approach.

Confined gradual typing marks an alternate approach for managing the performance of gradual typing via explicit annotations [Allende et al. 2014]. These annotations can be used to control the flow of type information, forbidding boundary crossings that the programmer suspect might cause significant slowdown. While our approach implicitly recovers and synchronizes type information to eliminate needless casts, there a programmer explicitly prevents expensive interactions ahead of time. The impact of using two different versions of a function for applications in different contexts on performance was explored by Allende et al. [2013] (without designing an optimization approach). They discovered that certain calls performed better when casts on arguments were placed inside a function (like in transient semantics) while other calls performed better when casts are directly

placed on arguments at the call site. DT removes casts in the body of the fast version function and also those at the call site if the type of the argument is identical to the fast version's parameter type.

Changes to cast semantics have been explored to bring some notion of improvement over the standard blame calculus [Feltey et al. 2018a; Garcia 2013; Herman et al. 2010; Siek et al. 2015a,c; Siek and Wadler 2010]. Nevertheless, not all casts can be eliminated in these approaches. Thus, our approach is complementary to them in that DT can optimize certain casts away in these approaches as we did for the guarded semantics. Several approaches have explored changing runtime environment using JIT speculative optimization techniques for improving gradual typing performance [Bauman et al. 2017; Richards et al. 2017]. Since DT is agnostic to implementation techniques, it seems that DT can be employed to further improve the performance of such approaches, for example, for eliminating higher-order casts that are challenging for JIT [Richards et al. 2017]. On the other hand, such approaches could also improve the performance of DT–indicating a synergistic relationship. Our result with PyPy is evidence of this.

The work on Grift [Kuhlenschmidt et al. 2019] explored the idea of implementing coercions and compiling gradual programs directly to assembly for a small formal calculus. In Grift, certain mixes of type annotations can still cause significant overhead compared to untyped programs. The authors conjectured that type information could be used to improve the performance of Grift, and in this paper we give an affirmative answer to that conjecture.

The main difference between our approach and those in this section is that ours does not change the source or target languages, the cast semantics, or the underlying language implementation.

Code specialization Code specialization has been widely explored to improve runtime performance in many domains, including embedded systems [Consel and Noël 1996], parallel computing [Khan et al. 2008], dynamic languages [Bolz et al. 2011, 2009; Chambers 1992; Gal et al. 2009; Kedlaya et al. 2013], and object-oriented programming [Cooper et al. 1992; Dean et al. 1995].

Code specialization can be static, dynamic, or hybrid. The main issue with static specialization is that the compiled program may have an explosive code size. In contrast, our approach increases code size by at most a factor of 2. Dynamic specialization [Consel and Noël 1996; Poletto et al. 1999] may incur extra overhead of compiling and generating code at runtime. JIT can be considered as a dynamic specialization technique, and our evaluation shows that DT and JIT are complementary.

Hybrid specialization [Carvalho et al. 2015; Khan et al. 2008] aims to address the shortcomings of both static and dynamic specializations by generating a limited number of templates for a function at compile time and performing efficient instantiation at runtime. The main challenges in hybrid specialization are generating effective templates and choosing an appropriate template to instantiate, which are very different from our challenges of finding optimizable parameters and deciding appropriate version of the function to use.

As gradual typing is often obtained by adding static types to dynamic languages, our approach is most closely related to dynamic specialization. At the behavioral level, as we said earlier, DT and dynamic specialization are complementary. First, the translation result of DT is still often run as a dynamic program, whose performance could be improved under systems that support code specialization. Second, it is unlikely that dynamic type specialization will eliminate inserted casts. For a cast inserted program without optimization, dynamic code specialization may create specialized versions when the arguments are given. Such specializations, however, could not eliminate casts inserted into the program. To some degree, we can say that DT optimizes programs at an early stage using domain knowledge, while dynamic code specialization optimizes at a later stage using profiled type information.

DT and dynamic code specialization also have many differences at the technical level. First, DT infers the type of parameters statically while dynamic code specialization obtains type information

through profiling at runtime. Second, dynamic code specialization often requires type guards to decide whether the specialized code can be used to handle later iterations or function calls. Such type guards are unnecessary in DT. Third, code specialization often happens at the function/method or trace (loop iteration) level. In contrast, DT optimization may happen at any level, as fine-grained as single function calls. Fourth, in dynamic code specialization, compilation of functions or traces often happen at runtime whereas in DT the translation is performed at compile time, during cast insertion.

Ortin et al. [2019] developed a code specialization based approach for optimizing gradual typing performance. Their approach has several important differences from our approach. First, their approach may change the behavior of the original program in that it does not preserve the blaming locations, which is an important design goal of our approach. Second, their approach generates specialized versions of a function based on the argument types, essentially generating a version for each type of the argument. Instead, we generate fast versions based on how parameters are used in the function definition, regardless of the argument types, which only decide whether the fast or normal version of the function will be used to perform the call.

Reusing the typing process DT employed the idea of variational typing [Chen et al. 2014] to reuse the typing process for finding optimizable parameters and and best translations. The work by Agesen [1995] developed a similar idea of reusing typing based on Cartesian Products (CPA) of types of arguments to a function. The main difference is that CPA works at the function/method level while variational typing works at expression level. In CPA, each unique element of the Cartesian product requires typing the method once. In contrast, in variational typing, the whole method is typed just once as parameter types are assigned variational types.

9 CONCLUSION

To combat the performance overhead incurred by sound gradual typing, we have presented discriminative typing–an optimization approach that works by compiling fast and slow versions of function definitions. The key insight is that by storing an extra version of functions based on an aggressive but safe inference, discriminative typing makes more optimizations than previous inference based approaches. Subtyping determines when it is safe to call the fast version of functions, guaranteeing that optimizations never change the behavior of programs. We evaluate discriminative typing on both Reticulated Python and Grift and our results demonstrate that it significantly improves performance of them. We therefore believe that DT is viable for improving gradual typing performance and is orthogonal to other approaches.

ACKNOWLEDGMENTS

We thank the anonymous reviewers from OOPSLA and POPL. Their reviews have significantly improved both the content and the presentation of this paper. This work is partially support by the National Science Foundation under the grant number CCF-1750886.

DATA AVAILABILITY STATEMENT

The proofs and additional evaluation figures are given in the appendices of the companion technical report [Campora et al. 2023]. A link to the project repository that contains prototype implementations as well as evaluated benchmarks and configurations is available on the author's website.

REFERENCES

Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 2–26. https://doi.org/10.1007/3-540-49538-X_2

Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. 2014. Confined Gradual Typing. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. ACM, New York, NY, USA, 251–270. https://doi.org/10.1145/2660193.2660222

Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-Typed Objects. In *Proceedings of the 9th ACM Dynamic Languages Symposium (DLS 2013)*. ACM Press, Indianapolis, IN, USA, 27–36. https://doi.org/10.1145/2508168.2508171 ACM SIGPLAN Notices , 49(2).

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2016. *Feature-Oriented Software Product Lines*. Springer.

Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133878

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C♯. In *ECOOP 2010 - Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–100. https://doi.org/10.1007/978-3-642-14107-2_5

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. 2011. Runtime Feedback in a Meta-Tracing JIT for Efficient Dynamic Languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Lancaster, United Kingdom) *(ICOOOLPS '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 8 pages. https://doi.org/10.1145/2069172.2069181

Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-Level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy) *(ICOOOLPS '09)*. Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018b. Migrating Gradual Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, CA, USA) *(POPL '18)*. ACM, New York, NY, USA. https://doi.org/10.1145/3158103

John Peter Campora, Sheng Chen, and Eric Walkingshaw. 2018a. Casts and Costs: Harmonizing Safety and Performance in Gradual Typing. *Proc. ACM Program. Lang.* 2, ICFP, Article 98 (July 2018), 30 pages. https://doi.org/10.1145/3236793

John Peter Campora, Mohammad Wahiduzzaman Khan, and Sheng Chen. 2023. Type-based Gradual Typing Performance Optimization. (2023). Available at https://people.cmix.louisiana.edu/schen/ws/techreport/dt-long.pdf.

Robert Cartwright and Mike Fagan. 1991. Soft Typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. ACM, New York, NY, USA, 278–292. https://doi.org/10.1145/113445.113469

Tiago Carvalho, Pedro Pinto, and João M. P. Cardoso. 2015. Programming Strategies for Contextual Runtime Specialization. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems* (Sankt Goar, Germany) *(SCOPES '15)*. ACM, New York, NY, USA, 3–11. https://doi.org/10.1145/2764967.2764973

Craig David Chambers. 1992. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. Ph. D. Dissertation. Stanford, CA, USA. UMI Order No. GAX92-21602.

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2012. An Error-tolerant Type System for Variational Lambda Calculus. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. ACM, New York, NY, USA, 29–40. https://doi.org/10.1145/2364527.2364535

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2014. Extending Type Inference to Variational Programs. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 1 (March 2014), 54 pages. https://doi.org/10.1145/2518190

Charles Consel and François Noël. 1996. A General Approach for Run-time Specialization and Its Application to C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. ACM, New York, NY, USA, 145–156. https://doi.org/10.1145/237721.237767

K.D. Cooper, M.W. Hall, and K. Kennedy. 1992. Procedure cloning. In *Proceedings of the 1992 International Conference on Computer Languages*. 96–105. https://doi.org/10.1109/ICCL.1992.185472

Jeffrey Dean, Craig Chambers, and David Grove. 1995. Selective Specialization for Object-Oriented Languages. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/207110.207119

Martin Erwig and Eric Walkingshaw. 2011. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Softw. Eng. Methodol.* 21, 1, Article 6 (Dec. 2011), 27 pages. https://doi.org/10.1145/2063239.2063245

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018a. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 133 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276503

Daniel Feltey, Ben Greenman, Christophe Scholliers, Robert Bruce Findler, and Vincent St-Amour. 2018b. Collapsible Contracts: Fixing a Pathology of Gradual Typing. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 133 (oct 2018), 27 pages. https://doi.org/10.1145/3276503

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

Ronald Garcia. 2013. Calculating Threesomes, with Blame. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. ACM, New York, NY, USA, 417–428. https://doi.org/10.1145/2500365.2500603

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Ben Greenman and Matthias Felleisen. 2018. A Spectrum of Type Soundness and Performance. *Proc. ACM Program. Lang.* 2, ICFP, Article 71 (July 2018), 32 pages. https://doi.org/10.1145/3236766

Ben Greenman and Zeina Migeed. 2018. On the Cost of Type-tag Soundness. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation* (Los Angeles, CA, USA) *(PEPM '18)*. ACM, New York, NY, USA, 30–39. https://doi.org/10.1145/3162066

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 (2019), e4. https://doi.org/10.1017/S0956796818000217

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. In *Selected Papers of the Symposium on Fourth European Symposium on Programming* (Rennes, France) *(ESOP'92)*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 197–230. http://dl.acm.org/citation.cfm?id=197475.190867

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient Gradual Typing. *Higher Order Symbol. Comput.* 23, 2 (June 2010), 167–189. https://doi.org/10.1007/s10990-011-9066-z

Jian Huang, Michael Allen-Bond, and Xuechen Zhang. 2017. Pallas: Semantic-Aware Checking for Finding Deep Bugs in Fast Path. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 709–722. https://doi.org/10.1145/3037697.3037743

Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved Type Specialization for Dynamic Scripting Languages. *SIGPLAN Not.* 49, 2 (oct 2013), 37–48. https://doi.org/10.1145/2578856.2508177

K. Kelsey, T. Bai, C. Ding, and C. Zhang. 2009. Fast Track: A Software System for Speculative Program Optimization. In *2009 International Symposium on Code Generation and Optimization*. 157–168. https://doi.org/10.1109/CGO.2009.18

Minhaj Ahmad Khan, H. P. Charles, and D. Barthou. 2008. An Effective Automated Approach to Specialization of Code. In *Languages and Compilers for Parallel Computing*, Vikram Adve, María Jesús Garzarán, and Paul Petersen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 308–322. https://doi.org/10.1007/978-3-540-85261-2_21

Alex Kogan and Erez Petrank. 2012. A Methodology for Creating Fast Wait-free Data Structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/2145816.2145835

Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2019. Toward Efficient Gradual Typing for Structural Types via Coercions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 517–532. https://doi.org/10.1145/3314221.3314627

Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. 2001. Specialization Tools and Techniques for Systematic Optimization of System Software. *ACM Trans. Comput. Syst.* 19, 2 (May 2001), 217–251. https://doi.org/10.1145/377769.377778

Philippe Meunier, Robert Bruce Findler, and Matthias Felleisen. 2006. Modular Set-based Analysis from Contracts. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. ACM, New York, NY, USA, 218–231. https://doi.org/10.1145/1111037.1111057

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

Colin Moock. 2004. *Essential ActionScript 2.0.* O'Reilly Media, Inc.

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

Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. In *OOPSLA* (Vancouver, British Columbia, Canada). ACM, New York, NY, USA. https://doi.org/10.1145/3133880

Phúc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2017. Soft Contract Verification for Higher-order Stateful Programs. *Proc. ACM Program. Lang.* 2, POPL, Article 51 (Dec. 2017), 30 pages. https://doi.org/10.1145/3158139

Phúc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. 2014. Soft Contract Verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM, New York, NY, USA, 139–152. https://doi.org/10.1145/2628136.2628156

Francisco Ortin, Miguel Garcia, and Seán McSweeney. 2019. Rule-based program specialization to optimize gradually typed code. *Knowledge-Based Systems* 179 (2019), 145–173. https://doi.org/10.1016/j.knosys.2019.05.013

Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 1999. C and Tcc: A Language and Compiler for Dynamic Code Generation. *ACM Trans. Program. Lang. Syst.* 21, 2 (March 1999), 324–369. https://doi.org/10.1145/316686.316697

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. ACM, New York, NY, USA, 481–494. https://doi.org/10.1145/2103656.2103714

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133879

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100. https://doi.org/10.4230/LIPIcs.ECOOP.2015.76

Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 425–435. https://doi.org/10.1145/2737924.2737968

Jeremy Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015c. Monotonic References for Efficient Gradual Typing. https://doi.org/10.1007/978-3-662-46669-8_18

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *In Scheme and Functional Programming Workshop*. 81–92. https://doi.org/10.1007/978-3-540-73589-2_2

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 32)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 274–293. https://doi.org/10.4230/LIPIcs.SNAPL.2015.274

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and Without Blame. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/1706299.1706342

Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S. New, Jan Vitek, and Matthias Felleisen. 2016. Is Sound Gradual Typing Dead?. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. ACM, New York, NY, USA, 395–406. https://doi.org/10.1145/1328438.1328486

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '14)*. ACM, New York, NY, USA, 45–56. https://doi.org/10.1145/2661088.2661101

Michael M. Vitousek, Jeremy G. Siek, and Avik Chaudhuri. 2019. Optimizing and Evaluating Transient Gradual Typing. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (Athens, Greece) *(DLS 2019)*. ACM, New York, NY, USA, 28–41. https://doi.org/10.1145/3359619.3359742

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 *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(ESOP '09)*. Springer-Verlag, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-

642-00590-9_1

Wen Xu, Sanjeev Kumar, and Kai Li. 2004. Fast Paths in Concurrent Programs. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*. IEEE Computer Society, Washington, DC, USA, 189–200. https://doi.org/10.1109/PACT.2004.16

Received 2023-07-11; accepted 2023-11-07