Casts and Costs: Harmonizing Safety and Performance in Gradual Typing

JOHN PETER CAMPORA, University of Louisiana at Lafayette, USA
SHENG CHEN, University of Louisiana at Lafayette, USA
ERIC WALKINGSHAW, Oregon State University, USA

Gradual typing allows programmers to use both static and dynamic typing in a single program. However, a well-known problem with sound gradual typing is that the interactions between static and dynamic code can cause significant performance degradation. These performance pitfalls are hard to predict and resolve, and discourage users from using gradual typing features. For example, when migrating to a more statically typed program, often adding a type annotation will trigger a slowdown that can be resolved by adding more annotations elsewhere, but since it is not clear where the additional annotations must be added, the easier solution is to simply remove the annotation.

To address these problems, we develop: (1) a static cost semantics that accurately predicts the overhead of static-dynamic interactions in a gradually typed program, (2) a technique for efficiently inferring such costs for all combinations of inferrable type assignments in a program, and (3) a method for translating the results of this analysis into specific recommendations and explanations that can help programmers understand, debug, and optimize the performance of gradually typed programs. We have implemented our approach in Herder, a tool for statically analyzing the performance of different typing configurations for Reticulated Python programs. An evaluation on 15 Python programs shows that Herder can use this analysis to accurately and efficiently recommend type assignments that optimize the performance of these programs without sacrificing the safety guarantees provided by static typing.

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

Additional Key Words and Phrases: gradual typing, variational types, cast insertion, cost analysis

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

1 INTRODUCTION

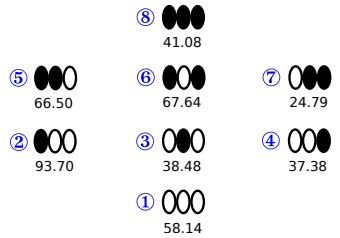
Static and dynamic typing have different strengths and weaknesses. Gradual typing [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006] attempts to combine the strengths of both by allowing programmers to statically type some parts of their program while dynamically typing other parts. Ideally, programmers could easily migrate between more or less statically typed programs by adding or removing type annotations. Intuitively, more static programs might be expected to have better performance and reliability, while more dynamic programs are more flexible and can be executed even if they are statically ill-typed. Unfortunately, migrating gradually typed

Authors' addresses: John Peter Campora, CACS, University of Louisiana at Lafayette, USA, campora@louisiana.edu; Sheng Chen, CACS, University of Louisiana at Lafayette, USA, chen@louisiana.edu; Eric Walkingshaw, School of EECS, Oregon State University, USA, walkiner@oregonstate.edu.

2475-1421/2018/9-ART98
https://doi.org/10.1145/3236793

```
def bench_spectral_norm(loops):
    range_it = xrange(loops)
    for _ in range_it:                          l3
        u = [1.0] * DEFAULT_N
        for x in xrange(10):                    l4
            part_A_times_u((x,u))
            part_At_times_u((x,u))
def eval_A(i, j):
    return 1.0 / ((i + j) * (i + j + 1) // 2 + i + 1)
def part_A_times_u(i_u):
    i, u = i_u
    partial_sum = 0.0
    for j, u_j in enumerate(u):                 l1
        partial_sum += eval_A(i, j) * u_j
    return partial_sum
def part_At_times_u(i_u):
    i, u = i_u
    partial_sum = 0.0
    for j, u_j in enumerate(u):                 l2
        partial_sum += eval_A(j, i) * u_j
    return partial_sum
```

(a) A program adapted from Python Benchmark Suite. The functions `part_A_times_u` and `part_At_times_u` are identical except for the underlined parts. The loop labels l_1, l_2, l_3, and l_4 are referenced in the cost lattice in Figure 2.



(b) Each node indicates whether the `eval_A`, `part_A_times_u`, and `part_At_times_u` functions are annotated (filled oval) or not (unfilled).

Fig. 1. A Python program (left) and its performance lattice (right)

programs is difficult [Campora et al. 2018; Tobin-Hochstadt et al. 2017] and can cause reliability and performance issues [Allende et al. 2014]. In particular, Takikawa et al. [2016] observed that adding type annotations can cause a more than 100 times slowdown in Typed Racket.

1.1 Performance Problem of Gradual Typing

To illustrate the performance implications of migrating between gradually typed programs, consider the program in Figure 1(a) for computing spectral norms of matrices, which was adapted from the Python Benchmark Suite.[1] The loop labels (l_1, l_2, l_3, and l_4) in the figure can be ignored for now. *Reticulated Python* [Vitousek et al. 2014] is a gradually typed variant of Python, where type annotations can be added, using the Python type hints syntax [van Rossum et al. 2014], to introduce static checking. For example, we could add type annotations to the `eval_A` function as shown below.

```
def eval_A(i:float, j:float)->float
```

In general, we can separately decide whether or not to annotate each of the four functions in the program, yielding $2^4 = 16$ potential typing *configurations*. Perhaps surprisingly, the choice of which functions to annotate has a significant and non-monotonic impact on the performance of the program. In Figure 1(b), we illustrate this with a lattice that shows the execution time[2] in Reticulated for each of the 8 configurations where `bench_spectral_norm` is left unannotated. Each node in the

[1] https://github.com/python/performance/blob/master/performance/benchmarks/bm_spectral_norm.py

[2] All times in this paper are in seconds and are measured on a laptop with 4 GB of RAM and an AMD A6-3400M quad-core processor running 64 bit Fedora 23.

lattice indicates whether the functions `eval_A`, `part_A_times_u`, and `part_At_times_u` are annotated (filled) or not (unfilled). For example, the node ① represents the configuration where no functions are annotated, while node ② represents the configuration where only `eval_A` is annotated.

Figure 1(b) shows that as we migrate the program to be more static (move up the lattice), the change in performance is unpredictable. For example, following the path ① → ② → ⑤ → ⑧, we see the performance first decreases, then increases twice. On the other hand, performance in ① → ④ → ⑦ → ⑧ first increases twice, then decreases. Additionally, the execution times at different configurations are very different, for example, the execution time at ② is about 3 times more than at ⑦. Together, these phenomena make it difficult for programmers to reason about what configurations lead to acceptable performance. Moreover, sometimes programmers are presented with a dilemma of whether static type checking is worth the performance slowdown to their application.

The unpredictable and sometimes severely negative performance impact of adding type annotations makes programmers reluctant to add them, and so the potential benefits of gradual typing go unrealized. This has lead Takikawa et al. to raise the question, "Is sound gradual typing dead?" [Takikawa et al. 2016]. Of course, adding type annotations can also improve performance. In Figure 1(b), the performance at node ⑦ is much better than the original at node ①, in addition to providing increased safety from static checking. Similarly, while annotating one subset of modules led to the 100 times slowdown observed by Takikawa et al. in Typed Racket, annotating a different subset of modules improved performance [Takikawa et al. 2016]. The crux of the problem is that it is not clear in advance what annotations the programmer should or should not add in order to both increase static type safety and maximize performance.

1.2 Program Migration Scenarios

Programmers are reluctant to use gradual typing features due to the difficulty of predicting the performance impact of type annotations, as described in Section 1.1. The goal of our work is to remove this barrier by providing tooling that helps programmers understand and reason about the tradeoffs between the safety guarantees given by increased static type checking and performance during program migrations. In this subsection we enumerate four scenarios that such tooling should support.

(S1) *Maximizing static typing.* In this scenario, the programmer's primary goal is to maximize the amount of static type checking, while performance is a secondary concern. Maximizing static checking typically entails adding as many type annotations as possible. However, often the most static migration of a gradually typed program is not unique, and so the programmer wants to pick the most performant migration amongst the set of possibilities. The following program, adapted from [Campora et al. 2018], illustrates the non-uniqueness of most-static migrations. In this example, a type annotation may be added to the parameter `fixed` or to `widthFunc`, but not to both.

```
def width(fixed, widthFunc):
  if (fixed):
    widthFunc(fixed)
  else:
    widthFunc(5)
```

Campora et al. [2018] reported hundreds of different ways to maximize static typing in larger programs. Since each of the most-static migrations may have significantly different performance profiles, it is important that the programmer can make a rational selection among them.

(S2) *Maximizing performance.* In this scenario, the primary goal is to maximize performance, while increasing static type checking is a secondary concern. Therefore, the programmer needs support locating places to add type annotations that will either improve or at least not degrade

performance. For example, starting from configuration ① in Figure 1, our tool should recommend configuration ⑦, since it has the least running time. This scenario can be extended in two ways: First, the programmer may want to maximize performance while providing type annotations in key places where increased static checking is judged to be more important or beneficial. For example, assume the function `eval_A` in Figure 1 must be annotated. Then the tool should consider configurations ②, ⑤, ⑥, and ⑧, and suggest configuration ⑧ as the most performant. Second, the programmer way want to maximize performance while restricting the number of proposed type annotations in order to manage the migration in a more incremental way. For example, starting from ①, if the programmer wants to add only a single annotation, then configurations ③ and ④ are preferable to ②.

(S3) *Increasing static type information without sacrificing performance.* In this scenario, the programmer wants to increase the static checking present the program, but only if it does not decrease its performance. To support this scenario, the tool should be able to identify type annotations that can be added that do not incur a performance overhead. For example, starting from configuration ①, the tool might recommend migrations ③, ④, ⑦, or ⑧, all of which increase the amount of static typing without sacrificing performance. Additionally, if a previous migration hurt performance, the tool should be able to identify subsequent migrations to improve it again. For example, migrating from ① to ② significantly decreases performance, but the tool should be able to recommend ⑧ as migration that will further increase safety guarantees from increased static typing while restoring performance to (better than) previous levels.

(S4) *Explaining performance degradation.* In this scenario, the programmer has experienced a performance degradation after adding type annotations and wants to understand why. Or, alternatively, the programmer wants to understand why the tool recommends against a particular migration. To support this, the tool should not only identify which program migrations will perform poorly, but provide an *explanation* of why this performance degradation occurs. For example, when the tool recommends against a migration from ① to ②, it might also explain that ② contains expensive type casts in a deeply nested loop. Such explanations help the programmer to develop their own mental model of how gradual typing affects the performance of their program. This enables them to use gradual typing features more effectively, and to make more informed program migrations.

In each of these scenarios, we assume the programmer wants to make decisions with respect to performance without decreasing the amount of typing. That is, we assume the tool will not recommend the removal of type annotations. The approach described in this paper follows this assumption, but extending it to also support the removal of type annotations poses no fundamental difficulties.

1.3 Capabilities of a Tool to Support Program Migration

Each of the scenarios in Section 1.2 involve exploring many alternative configurations of a gradually typed program. Without tool support, this is extremely tedious since it requires manually adding and removing type annotations and rerunning the program to measure its performance. Worse, this exploration cannot hope to be complete for large programs since the search space is simply too large, so the best configuration for the scenario will not likely be found. Therefore, tool support is necessary to support the scenarios and to help programmers effectively migrate gradually typed programs. This subsection identifies the key capabilities needed to build such a tool, and outlines the techniques we use to provide them.

We propose a methodology for systematically exploring the entire space of potential program migrations needed to support each scenario and to efficiently identify the most performant type configurations in that space. We can break this methodology down into three fundamental capabilities: (C1) A way to enumerate and efficiently represent all valid type configurations of a gradually

typed program. (C2) A way to statically approximate and compare the runtime performance of a single configuration of a gradually typed program. (C3) A way to combine C1 and C2 to efficiently compute and compare the performance approximations for all type configurations.

Regarding C1, in a program with n parameters, an upper limit on the number of possible type configurations is 2^n since each parameter can either be assigned a static type or remain unannotated (and thus dynamically typed). However, in general, not every combination of parameters can be statically annotated in a consistent way, as illustrated by the `width` example in Section 1.2. We can use *variational type inference* [Chen et al. 2014] to efficiently explore all 2^n possibilities by inferring static types for all parameters in one pass while keeping track of which combinations of types are compatible with each other. The idea to use type inference to help migrate gradually typed programs is inspired by the observation that "static type systems accommodate common untyped programming idioms" by Takikawa et al. [2016], and by previous successes combining gradual typing and type inference [Campora et al. 2018; Garcia and Cimini 2015; Rastogi et al. 2012; Siek and Vachharajani 2008]. In particular, we reuse the machinery developed in Campora et al. [2018] to transform the output of variational type inference into an efficient representation of all valid type configurations of a program.

As in previous work, the success of type inference in a gradually typed setting can be expected to vary significantly across programs. In our benchmarks, we observed a broad range of outcomes, successfully inferring types for anywhere from 25% to 100% of parameters in a program (Section 6.2). Fortunately, effectively supporting the migration scenarios does not require inferring types for all parameters. We simply infer types for as many parameters as possible, then reason about this space of migrations. Subsequent migrations may require fundamental changes to the code to remove behavior that relies on dynamic typing, expanding the space we can reason about.

Capability C2 requires a way to estimate the overhead of gradual typing in each configuration, allowing us to estimate and rank their expected runtime performance. To enable C2, we develop a static *cost semantics* for gradually typed programs. The insight underlying our cost semantics is that the overhead of gradual typing is mostly caused by inserted casts [Takikawa et al. 2016] and checks [Vitousek et al. 2014, 2017]. Therefore, we statically approximate the number and complexity of cast and check operations that will be performed while executing a gradually typed program. Figure 2 shows the result of applying our cost semantics to each of the configurations of the program in Figure 1. Since we do not know statically how many times each loop body in the program will be executed, the costs are parameterized by symbolic values representing the number of iterations (l_1, l_2, l_3, and l_4).

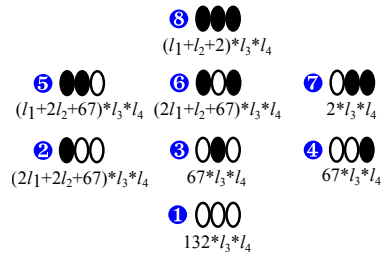


Fig. 2. Cost lattice for the program in Figure 1. We have omitted an addend 2 that is shared by all configurations. The letters l_1, l_2, l_3, and l_4, introduced in Figure 1(a), represent the number of iterations for the respective `for` loops.

Note that our cost semantics does not approximate the absolute runtime of different configurations of the program, but only the overhead of gradual typing. This is similar to other cost analyses focused on specific aspects of program execution [Hoffmann and Hofmann 2010; Hoffmann and Shao 2015]. For example, the absence of loop labels l_1 and l_2 in ❶ does not suggest that these loops are not executed, but rather that no casts or checks will be performed in these loops in the corresponding configuration. The factors 2, 67, and 132 that multiply these loop labels correspond to the estimated overhead of individual casts and checks performed in the body of these loops.

Since gradual typing overhead is only a (smaller or larger) fraction of the running time of a program, we can see that the ratios of estimated costs for two configurations do not correspond

precisely to the ratios of their running time. For example, the cost of ❶ in Figure 2 is 66 times that of ❼, while the running time of ① in Figure 1(b) is roughly 2.5 times that of ⑦. However, the relative ordering of the costs in Figure 2 do correspond to the relative ordering of running times in Figure 2. For example, the cost at ❼ is $2 * l_3 * l_4$ is the lowest estimated cost and corresponds to the lowest running time at ⑦. Similarly, the cost decreases along the path ❶ → ❹ → ❼, just as the running time decreases along the path ① → ④ → ⑦. This illustrates that the estimated costs are useful as a tool for predicting the relative performance of different typing configurations. This makes sense since the different configurations differ only in their type assignments, and so any difference in their running time should be explained by the overhead of gradual typing.

One may ask how can we compare, for example, the costs of ❶ and ❽ since one mentions l_1 and l_2 and the other does not? The answer is: we can not. This makes sense because the relation of the running times between these two variants is not fixed. In this case, it depends on the value of `DEFAULT_N` (see Figure 1(a)), which determines how many times the loop bodies of l_1 and l_2 are executed. In general, different loop bodies can be expected to execute different numbers of times based on different inputs and environment settings. The loop labels make this uncertainty explicit. While costs involving different loop labels cannot be directly compared, they can still be used to produce explanations to help a programmer make an informed decision when deciding between different migrations. For example, the tool might explain that the configuration at ⑧ induces casts in the loops l_1 and l_2, but reduces the cost of casts in the loops l_3 and l_4, relative to configuration ①.

With capabilities C1 and C2, we can statically enumerate all type configurations of a program and statically compare the performance of different configurations using our cost semantics. Hypothetically, we could apply these together in order to identify the best configurations to support our program migration scenarios. The problem is that, since the number of type configurations produced by C1 scales exponentially with the number of parameters, the search space quickly grows overwhelming to perform the search by applying C2 directly. This problem is solved by C3, which enables efficiently computing and comparing costs for all valid type configurations.

The key observation to support C3 is that a substantial amount of work can be reused between the cost analyses of different configurations. For example, calculating the cost of configurations ❶ and ❸ can share the computations of costs for `eval_A` and `part_At_times_u`. Similarly, calculating the costs of ❸ and ❹ can share the computations of `eval_A`. In large programs, the majority of the computations can be shared when computing the costs of two similar configurations.

Unfortunately, sharing cannot be achieved by simply analyzing the costs of different functions separately then composing the results since interactions between functions do affect the analyses. However, by locally capturing differences between sets of configurations and preserving these differences throughout the analysis, we can effectively reuse results wherever possible. Specifically, we apply the ideas of *variational programming* [Chen et al. 2016; Erwig and Walkingshaw 2013] and *variational typing* [Chen et al. 2014] to systematically reuse computations during the cost analysis. Instead of enumerating all configurations and computing the cost of each separately, we perform a *variational cost analysis* that analyzes the program once to compute a *variational cost* that compactly represents the cost of all valid type configurations.

With these three capabilities, we can support all of the scenarios outlined in Section 1.3 by performing a variational cost analysis, then querying the variational cost to identify the desired configuration. For example, to support S1 (maximizing static typing), we would select the lowest cost among the configurations that include as many annotations as possible.

1.4 Relation with Previous Work and Contributions of this Work

There have been several lines of research addressing the performance problem of gradual typing since Takikawa et al.'s 2016 report on the prohibitive overhead of sound gradual typing. Previous

work has addressed the problem through new languages with more efficient gradually typed semantics [Muehlboeck and Tate 2017; Vitousek et al. 2017] and through new implementation techniques for existing languages [Bauman et al. 2017; Richards et al. 2017]. There are two main differences between our work and previous efforts: First, our approach does not require changes to existing gradually typed languages or implementations; it works with the prevailing implementation technique of translating a typed variant of a language into an underlying untyped language. Second, in addition to addressing the common goal of reducing or avoiding performance problems, our work also provides a way to understand and debug performance problems when they occur. We discuss the relation with existing work in more detail in Section 7.2.

By drawing insights from gradual typing and type inference [Campora et al. 2018; Garcia and Cimini 2015; Siek and Vachharajani 2008], cost analysis [Danner et al. 2015; Hoffmann and Hofmann 2010], and variational programming [Chen et al. 2012, 2014], we develop a methodology for understanding, debugging, and optimizing the performance of gradual programs based on a deep understanding of how types affect performance. To test the feasibility of this methodology, we have implemented our variational cost analysis as Herder, a tool that efficiently and accurately analyzes the costs of many type configurations of a Reticulated Python program. Overall, this paper makes the following contributions:

(1) We develop a cost semantics for casts in gradually typed programs with a guarded semantics in Section 4. The cost semantics is simple and enables automating cost analysis, yet still allows authentically comparing the relative run times of different configurations for the same program.
(2) We combine variations and cost analysis, yielding a variational cost analysis in Section 5. Instead of computing costs for all configurations separately, variational cost analysis systematically reuses computations to compute a variational cost that encodes the cost of all the configurations that can be inferred.
(3) We have implemented our approach as Herder on top of Reticulated and evaluate it in Section 6. We evaluate the accuracy of the cost semantics by taking the configuration Herder reports as having the lowest cost and testing whether it has the fastest runtime amongst the measured configurations. Our evaluation demonstrates that Herder can efficiently find configurations yielding good performance. In most benchmarks, the recommended configuration is one of the top 3 in terms of execution time. Moreover, our approach is scalable, taking exponentially less time than a brute-force approach as the number of configurations becomes large, and 2–4 times as long as the cost analysis of a single configuration.

The rest of the paper is organized as follows. Section 2 provides necessary background on gradual typing and variational typing. Section 3 informally introduces our static cost semantics, while Section 4 gives the formal definition. Section 5 gives the formal definition of the full variational cost semantics. The implementation of Herder is described in Section 6, together with an evaluation of its accuracy and scalability. Section 7 describes related work and Section 8 concludes.

2 TYPING, GRADUALLY AND VARIATIONALLY

This section provides background needed to understand the rest of the paper. In Section 2.1, we describe why and where casts are inserted into gradually typed programs. In Section 2.2, we review variational typing [Chen et al. 2012] as a way to efficiently analyze many variants of a program.

2.1 Gradual Typing

Gradual typing allows mixing dynamically typed and statically typed code within a single program. In a gradually typed program, statically typed values can flow into dynamically typed code and vice versa. For example, consider the following dynamically typed function `double`.

```
def double(x):
  return x * 2
```

Suppose the multiplication operation $*$ is statically typed as $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$. Then the dynamically typed argument x flows into this statically typed operation. Similarly, if we invoke this function as double(3), where 3 has static type Int, then a statically typed value flows into the dynamically typed function.

During static type checking, the interface between static and dynamic code is defined by a *consistency* relation [Garcia et al. 2016; Siek and Taha 2006]. Consistency (denoted by $\sim$) weakens type equality by making every type consistent with the type Dyn, representing the type of dynamically typed code. So the expression $\texttt{x} * \texttt{2}$ in the double function is statically type correct since x has type Dyn and $\texttt{Dyn} \sim \texttt{Int}$. Of course, in order to preserve the dynamic type safety of gradually typed programs, additional type checking may have to be performed at runtime. For example, the double function must be translated to a version with an explicit *type cast* $\lceil \texttt{Int} \Leftarrow \texttt{Dyn} \rceil$ as shown below.

```
def double(x):
  return ⌈Int ⇐ Dyn⌉x * 2
```

During program translation, such casts are inserted into the program wherever dynamically and statically typed code interact. Note that we do not need to cast the value 2 to Int since its type is statically known. Similarly, if we annotated the type of double to be $\texttt{Int} \to \texttt{Int}$, we would not need to cast x to Int, and the function call double(3) would not involve any casts since all types are statically known.

2.2 Variational Typing

Variational typing was developed by Chen et al. [2014] to provide types for *variational programs*. A variational program represents several related program variants using *choices* [Erwig and Walkingshaw 2011] to denote where the variants differ, as illustrated below.

$$\texttt{result = } B\langle \texttt{odd}, \texttt{double} \rangle \texttt{(3)} \qquad \text{(vprog)}$$

The variational program vprog contains a choice named B with *alternatives* odd and double. Two distinct programs can be generated by *selecting* the first or second alternative of choice B. For example, selecting the first alternative, denoted $\lfloor \texttt{vprog} \rfloor_{B.1}$, yields the program result = odd(3), while selecting $\lfloor \texttt{vprog} \rfloor_{B.2}$ yields result = double(3).

Choices with the same name in a variational program are synchronized, while choices with different names are independent. That is, $\lfloor e \rfloor_{d.i}$ selects the ith alternative of all choices named d in e. We call $d.i$ a *selector* and range over selectors with s. Obtaining a plain (non-variational) program from a variational program may require several selections. We call a set of selectors a *decision*, ranged over by δ, and generalize the notation of selection to decisions as $\lfloor e \rfloor_\delta$.

Variation in expressions naturally gives rise to variation in types. For example, the expression $B\langle \texttt{odd}, \texttt{double} \rangle$ can be assigned the type $B\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int} \rangle$. Variational type systems extend traditional type systems to accommodate choices at the expression and type level. Typing function applications is complicated by the fact that both the function and argument may be variational. To accommodate this, variational type systems are equipped with a type equivalence relation. Two types T_1 and T_2 are equivalent, denoted $T_1 \equiv T_2$ if $\lfloor T_1 \rfloor_\delta = \lfloor T_2 \rfloor_\delta$ for every decision δ. So, for example, $B\langle \texttt{Int} \to \texttt{Bool}, \texttt{Int} \to \texttt{Int} \rangle \equiv \texttt{Int} \to B\langle \texttt{Bool}, \texttt{Int} \rangle$ since both sides of the equivalence yield $\texttt{Int} \to \texttt{Bool}$ when selecting $B.1$ and both sides yield $\texttt{Int} \to \texttt{Int}$ when selecting $B.2$. Taking the right-hand side of this equivalence, it is easy to see that the function application $B\langle \texttt{odd}, \texttt{double} \rangle \texttt{(3)}$ is well typed and yields a result of type $B\langle \texttt{Bool}, \texttt{Int} \rangle$.

A crucial property of variational type systems is that *selection preserves typing*. That is, if $e : T$ then $\forall s. \lfloor e \rfloor_s : \lfloor T \rfloor_s$. Previous work showed how variational types enable efficiently reasoning

about all possible assignments of dynamic or static types to function parameters in gradually typed programs [Campora et al. 2018]. This enables efficiently migrating between gradually typed programs with different type assignments. In this work, we tackle the problem of estimating the performance overhead associated with different migrations for a gradual program.

3 THE WORKFLOW OF HERDER

Herder works by generating and reasoning about variational programs that represent all possible type configurations at once. We use the following program to illustrate how this works, including how variational casts are inserted and how variational costs are computed. We assume the + operator has the static type `Int` → `Int` → `Int`.

```
def add(x, y):
   return x + y
```

Casts are only inserted when passing dynamically typed values to statically typed code, which can occur when some parts of the program contain type annotations, when primitive operations are assigned static types by the language implementation, or when typed external code is called. We focus on how programmer-added type annotations change the behavior of cast insertion. Therefore, before we can reason about the costs of inserted casts arising via different annotations, we need to infer what combinations of type annotations can be added to the program.

Migrational typing by Campora et al. [2018] can efficiently infer types for all configurations of a program in a gradually typed language with type inference. It is only necessary for type inference to be sound (not complete), so migrational typing can be applied even to languages with features, such as subtyping, that prevent complete type inference.

For `add`, we can use migrational typing to infer that each parameter can independently have type `Dyn` or `Int` (note that an unannotated parameter is equivalent to one annotated by `Dyn`). This yields four potential configurations of `add`, which we can represent in a single single variational program, `addV`, with two independent choices.

```
def addV(x:B⟨Dyn, Int⟩, y:D⟨Dyn, Int⟩):
  return x + y
```

In this paper, we focus on the problem of reasoning about and comparing costs for different configurations and reuse necessary machinery from Campora et al. [2018] to get type information.

Instead of generating all configurations of `addV` and separately reasoning about the casts in each, we instead add *variational casts* to `addV` and reason about `addV` directly. In this case, we need to insert variational casts to ensure that the arguments to + have the type `Int` in each configuration. Specifically, we insert the cast $\lceil$`Int` $\Leftarrow B\langle$`Dyn, Int`$\rangle\rceil$ `x`, and similarly for `y`. This represents the cast $\lceil$`Int` $\Leftarrow$ `Dyn`$\rceil$`x` in $B.1$, where `x` has static type `Dyn`, and it represents the cast $\lceil$`Int` $\Leftarrow$ `Int`$\rceil$`x` in $B.2$, where `x` has static type `Int`. Since $\lceil$`Int` $\Leftarrow$ `Int`$\rceil$ is a no-op, we can transform the variational cast applied to `x` to $B\langle\lceil$`Int` $\Leftarrow$ `Dyn`$\rceil, \epsilon\rangle$ `x`, making it clear that no cast (ϵ) is performed in the $B.2$ case where passing `x` to + can be statically type checked.

After cast insertion and simplification, we obtain `addVC`:

```
def addVC(x:B⟨Dyn, Int⟩, y:D⟨Dyn, Int⟩):
  return B⟨⌈Int ⇐ Dyn⌉, ϵ⟩ x + D⟨⌈Int ⇐ Dyn⌉, ϵ⟩ y
```

Our next goal is to find a way to measure the overhead of the inserted casts, and more importantly to compare the overhead of different configurations. For this simple program, we can simply count the number of inserted casts. Therefore, we assign the *variational cost* $B\langle 1, 0\rangle$ to the cast $B\langle\lceil$`Int` $\Leftarrow$ `Dyn`$\rceil, \epsilon\rangle$ `x`, and the cost $D\langle 1, 0\rangle$ for casting `y`. The cost for `addV` is then $B\langle 1, 0\rangle + D\langle 1, 0\rangle$. Applying standard variational programming techniques [Erwig and Walkingshaw 2013], we can

reduce this cost to $B\langle D\langle 2, 1\rangle, D\langle 1, 0\rangle\rangle$, which captures the costs of all four configurations of the program.

We can obtain the cost of each configuration by selecting from the variational cost with the corresponding decision. For example, selecting with $\delta = \{B.1, D.2\}$ yields 1, corresponding to the single cast of `x` in the configuration of `addV` produced by selecting with the same decision δ. From the variational cost, we can see that the configuration of `add` that annotates both parameters with `Int` leads to the lowest cost.

Sections 4 and 5 present a formal treatment of this process. To separate concerns, we first present the cost semantics for a single program in Section 4, then a method that computes the costs for all valid configurations for the given program in Section 5.

To give a high-level view of the formalization, we present the connections between various relations and syntaxes in Figure 3. Moving down the right side of the figure illustrates the process of assigning costs to individual plain programs, while moving down the left side of the figure illustrates the variational cost analysis process. Plain cost analysis works as follows (starting from the upper right corner and moving down): the syntax e_s represents input gradual programs, which may contain type annotations; the type-directed transformation $\vdash_G$ inserts casts into e_s, erases type annotations, and generates e_t, which can be directly executed on the underlying interpreter; we then use $\searrow$ to further transform programs in e_t into programs in the same syntax but which are more amenable to cost analysis, and compute costs A with the $\vdash_C$ relation.

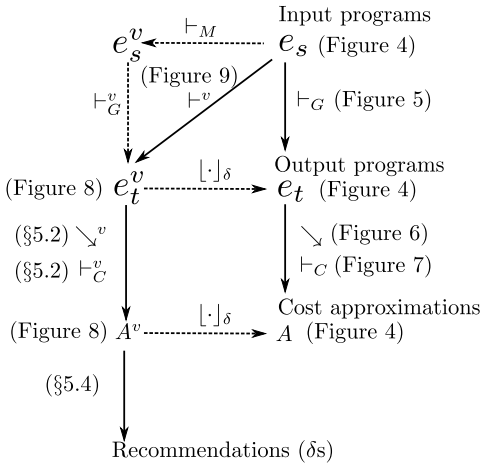


Fig. 3. Overview of computing costs for all valid type configurations. The operations and transformations attached to dashed arrows are not presented in this paper, they are either from previous work (such as $\vdash_M$ from Campora et al. [2018] and $\lfloor\cdot\rfloor_\delta$ from Chen et al. [2014]) or for aiding conceptual understanding ($\vdash_G^v$).

Conceptually, variational cost analysis works as follows (starting from the upper right corner, moving left, then down): starting from e_s, we apply migrational typing, $\vdash_M$, [Campora et al. 2018] to compute all valid type configurations, which we encode as e_s^v; we apply a variational transformation, $\vdash_G^v$, to insert variational casts yielding a variational version of the target language e_t^v; a variational transformation $\searrow^v$ makes the program more amenable to cost analysis; finally, we compute variational costs A^v with $\vdash_C^v$. In practice, however, we move directly from e_s to e_t^v using $\vdash^v$ since this simplifies the formalization. Since A^v encodes costs for all configurations, we can use this result to make recommendations to satisfy the current program migration scenario. The arrows annotated by $\lfloor\cdot\rfloor_\delta$ help establish the correctness of variational cost analysis by relating variational results with their corresponding plain results through selection (see Theorem 5.2).

4 COSTS FOR A SINGLE CONFIGURATION

Casts are the major source of performance degradation in gradually typed programs [Takikawa et al. 2016]. Therefore, we need a way to accurately estimate the costs associated with the inserted casts. In this section, we address this need by developing a static cost semantics in the style of Danner et al. [2015]. Our cost semantics produces an expression in a *cost language* that evaluates to a cost estimate for the corresponding gradually typed program. Specifically, in Section 4.1, we

define a function for estimating the cost of an individual cast. In Section 4.2, we introduce the cost language and cost semantics informally through an example, then give a formal treatment in Sections 4.3 through 4.5. In Section 4.6, we discuss important properties of the cost semantics.

4.1 The Cost of a Basic Cast

First, we consider the cost of an individual cast $\lceil G_1 \Leftarrow G_2 \rceil$, which checks that a value with gradual type G_2 can be converted into a value with gradual type G_1, and performs the conversion if necessary. The easiest cast to reason about is casts to `Dyn`. When $G_1 = \texttt{Dyn}$ and G_2 is a base type (such as `Int` or `Bool`) the cost of the cast is b, representing the cost of boxing a value.

The simplest cast that does some work is a cast where $G_2 = \texttt{Dyn}$ and G_1 is a base type. In this case, a dynamic type check must be performed. We use c to represent the constant cost of such basic casts. For casts involving list types in both the source and target, we use s to represent the overhead of checking the elements of a list, and we recursively compute the cost for the types inside the respective list constructors. Our cost function is below:

$$cost(\lceil G_1 \Leftarrow G_2 \rceil) = \begin{cases} b & G_1 = \texttt{Dyn} \\ s \cdot cost(\lceil G_1' \Leftarrow G_2' \rceil) & G_i = [G_i'] \\ c & \text{otherwise} \end{cases}$$

The *cost* function handles basic casts, which are simply performed at the locations they are encountered in the program. In contrast, casts involving function types or reference types are trickier since they are not applied immediately, but rather when the corresponding function or reference is used [Siek et al. 2009]. We illustrate how to account for the cost of such casts and give a precise cost calculation in Section 4.4.

4.2 Estimating Cast Costs in Programs

Armed with a cost function for individual casts, we can estimate the cost of a sequence of statements by simply summing up the costs of each. However, more interesting programs present two challenges: (1) How do we represent the costs of functions, whose bodies may contain casts whose individual costs depend on the types of the arguments they are applied to? (2) How do we estimate the cost of loops, whose bodies may contain casts that are executed a statically indeterminate number of times? In this subsection, we introduce a cost language and cost semantics for addressing these challenges. We use the following function `mult` as a running example.

```
def mult(md, mr):
  sum = 0
  for i in range(md):
    sum = add(sum, mr)
  return sum
```

This function multiplies the multiplicand `md` with the multiplier `mr` by iteratively adding `mr` to a local variable `sum`. The built-in Python function `range` : `Int` → `[Int]` returns a list `[1,2,...,n]` for the given `n`, and the helper function `add` returns the sum of its arguments.

First, we consider how to represent the cast cost of a function. A function by itself incurs no cost, but rather represents a *potential* cost when invoked. Moreover, the cast cost of the body of a function will vary depending on the arguments that are passed in. The potential cost of a function can be represented by a corresponding function in the cost language, and the cost of a function application can be approximated by executing a corresponding application in the cost language.

More concretely, following Danner et al. [2015], we represent the cost *approximation* of a term by a pair (C, P), where C represents the *immediate cost* of the term and P represents its *potential cost*.

We use A to range over approximations and use *cost* and *potential* to refer to immediate costs and potential costs, respectively. The potential of a function abstraction $\lambda x.e$ in the source language is captured by a *potential abstraction* in the cost language of the form $\Lambda x.A$. The cost of a function application $e_1\ e_2$ can then be computed by applying the corresponding potentials of e_1 and e_2.

Returning to our example, we can sketch a template for the approximation of `mult` as $(0, \Lambda md.(0, \Lambda mr.A_m))$, where the function itself (and its partial application) has no immediate cost, and the body of the function is approximated by A_m, which may refer to the potentials of its parameters, md and mr.

While we produce costs, we maintain a *cost environment* that maps each source language variable to its potential. Let us assume that `sum` has type `ref Dyn`. The first statement of the body is `sum = 0`, which has approximation $(0, 0)$. The cost is 0 since we generate no casts and 0 is a constant, and its potential is 0 since the statement does not involve functions or loops. In reality, `sum` is a reference and has associated potential. To handle this, we define a transformation in Section 4.4, but for simplicity while illustrating this example we will assume that reference creation and assignment contributes no overhead relating to proxies. Therefore, the statement contributes no cost to A_m and the environment is extended with $\texttt{sum} \mapsto 0$.

We now turn to producing a cost for the loop in `mult`. For the subexpression `range(md)`, suppose the cost environment maps the built-in function `range` to the potential $\Lambda u.(0, u)$, indicating that it incurs no cast costs except the potential costs of its argument. Since `md` has type `Dyn`, a cast $\lceil \texttt{Int} \Leftarrow \texttt{Dyn} \rceil$ must be performed before passing `md` into `range`. This cast has cost c. The overall approximation of `range(md)` is this immediate cost added to the approximation returned by applying the potential of `range` to the potential of the argument `md`, which is $(\Lambda u.(0, u))\ \ md = [md/u](0, u) = (0, md)$. This yields a final approximation of (c, md) for the subexpression `range(md)`.

The cost of a loop is the cost of its body times the number of iterations. In general, the number of iterations is unknown statically. Therefore, in our approximations we introduce a unique symbolic value to stand for the number of iterations each loop executes, which we call a *loop label*. The cost approximation of a loop is the cost and potential of its body, each multiplied by the loop label.

In `mult`, the body of the loop is `sum = add(sum, mr)`. For simplicity, let us assume for now that the assignment does not generate a reference cast and that the use of `sum` in the body (a dereference) does not generate a cast from a proxy. Now we can focus on the application of `add` to `sum` and `mr`. Suppose `add` is annotated as $\texttt{Int} \to \texttt{Int} \to \texttt{Int}$, then we must cast both arguments for an immediate cost of $2c$. Also suppose `add` has potential $\Lambda x.(0, \Lambda y.(c, x + y))$. As described above, the cost environment maps $\texttt{sum} \mapsto 0$ and $\texttt{mr} \mapsto mr$, so the resulting approximation of the application is $(c, 0 + mr) = (c, mr)$. Adding the immediate costs of the casts to this approximation yields an overall approximation of $(3c, mr)$ for the loop body. To approximate the overall cost of the loop, we multiply the cost of the body by a new loop label l, yielding $(3c \cdot l, mr \cdot l)$, then add the approximation of `range(md)`, yielding $(3c \cdot l + c, mr \cdot l + md)$.

Finally, the function returns a dereference of `sum`. This has cost $(0, 0)$ since the return type of `mult` is `Dyn`. Since the loop is the only source of cast costs in `mult`, we can set A_m in our template above to the loop approximation to yield our final approximation for the whole function: $(0, \Lambda md.(0, \Lambda mr.(3c \cdot l + c, mr \cdot l + md)))$.

4.3 Cast Insertion Rules

As the running example in Section 4.2 shows, our idea of computing costs is to measure the number and complexity of casts in programs. This subsection presents rules for inserting casts into gradually typed programs. In Figure 4, we define a simple calculus that captures the essential features of gradually typed programs with respect to cost analysis. The syntax of expressions is lambda calculus extended by references and a loop construct. The syntax of gradual types consists of base types (γ),

Term variables $\mathsf{x}, \mathsf{y}, \mathsf{z}$ Base types γ Cost variables x, y, z Loop labels l

$$
\begin{array}{lrcl}
\text{Source expr.} & e_s & ::= & \mathsf{x} \mid \lambda \mathsf{x} \colon G.e_s \mid e_s\ e_s \\
 & & \mid & \textbf{for}\ \mathsf{x}\ \textbf{in}\ e_s\ \textbf{do}\ e_s \\
 & & \mid & \textbf{let}\ \mathsf{x} = e_s\ \textbf{in}\ e_s \\
 & & \mid & \texttt{letrec}\ \mathsf{x} = e_s\ \texttt{in}\ e_s \\
 & & \mid & \texttt{ref}\ e_s \mid \texttt{!}e_s \mid e_s \texttt{:=} e_s \\
\text{Target expr.} & e_t & ::= & \mathsf{x} \mid \lambda \mathsf{x}.e_t \mid e_t\ e_t \\
 & & \mid & \textbf{for}\ \mathsf{x}\ \textbf{in}\ e_t\ \textbf{do}\ e_t \\
 & & \mid & \textbf{let}\ \mathsf{x} = e_t\ \textbf{in}\ e_t \\
 & & \mid & \texttt{letrec}\ \mathsf{x} = e_t\ \texttt{in}\ e_t \\
 & & \mid & \texttt{ref}\ e_t \mid \texttt{!}e_t \mid e_t \texttt{:=} e_t \\
 & & \mid & \lceil G \Leftarrow G \rceil e_t
\end{array}
$$

$$
\begin{array}{lrcl}
\text{Type constr.} & T & ::= & [] \mid \texttt{ref} \\
\text{Gradual types} & G & ::= & \gamma \mid T\ G \mid G \to G \mid \texttt{Dyn} \\
\\
\text{Approximations} & A & ::= & (C, P) \mid C \oplus P \\
\text{Costs} & C & ::= & n \mid l \cdot C \mid C + C \\
\text{Potentials} & P & ::= & n \mid x \mid l \cdot P \mid P + P \\
 & & \mid & \Lambda x.A \mid P\ P \\
\\
\text{Type env.} & \Gamma & ::= & \emptyset \mid \Gamma, \mathsf{x} \mapsto G \\
\text{Cost env.} & \Phi & ::= & \emptyset \mid \Phi, \mathsf{x} \mapsto P
\end{array}
$$

Fig. 4. Syntax for our gradually typed functional language and its corresponding cost language.

list types, function types, and the dynamic type. List types can be introduced through user type annotations or the initial type environment. The syntax of approximations, costs, and potentials are as described in Section 4.2. The n in the syntax refers to b, c, and s in Section 4.1. We discuss the syntax $C \oplus P$ in Section 4.5.

In Figure 5, we define the cast insertion procedure for this calculus as a part of the typing process. The judgment $\Gamma \vdash_G e_s \rightsquigarrow e_t : G$ can be read as: given a type environment Γ and a source expression e_s, e_s has type G and is translated to a target expression e_t, which contains inserted casts. The formalization is fairly standard, except for the addition of rule For.

We use the syntactic form $[\![G_2 \Leftarrow G_1]\!]$ to denote casts that can potentially be inserted, that is, they are inserted only when $G_1 \neq G_2$. The rule definitions use several helper functions, given at the bottom of Figure 5. These functions extract certain parts from types when they have desired structures or Dyn when they are Dyn. Otherwise, these functions are undefined. For example, the function ext_L extracts the element type from a list type and Dyn from Dyn. The helper functions allow us to create a single rule for each source language construct, regardless of types.

The Var rule for variable references is standard. No casts are inserted in the translation process. The Abs rule is also standard, except that it removes the parameter's annotation, since the target language is untyped. The App rule uses the consistency relation ($\sim$) to make sure that the type of the argument is consistent with the domain of the function. The definition of $\sim$ is standard [Siek and Taha 2006], and we omit it here. If the two types are consistent, the rule translates both expressions and inserts casts on each. The function (e_{1t}) is cast to have a function type with the original type's domain and codomain, using the *dom* and *cod* helper functions. The argument is also cast so that its type matches the domain of the new function type.

In the For rule, there is a variable x associated with the loop that ranges over the list produced by e_{1s}. Typing ensures that e_{1s} has a type that can be treated as a list by using the ext_L function. Consequently, the translation procedure inserts a cast on e_{1t} to a list type containing the extracted type. The Let and Letrec for non-recursive and recursive let-bindings are standard.

For references, the Ref rule is trivial since no casts are inserted. For dereferences, in the Deref rule, the source expression must be checked to ensure that its type is a reference via the ext_R function. Thus, the translation process inserts a cast to a reference of the type extracted by ext_R before dereferencing it. Finally, assignment in the Assign rule also applies ext_R to the reference

$$\textsc{Var}\ \frac{\mathtt{x} : G \in \Gamma}{\Gamma \vdash_G \mathtt{x} \rightsquigarrow \mathtt{x} : G} \qquad \textsc{Abs}\ \frac{\Gamma, \mathtt{x} \mapsto G \vdash_G e_s \rightsquigarrow e_t : G_1}{\Gamma \vdash_G \lambda \mathtt{x} : G.e_s \rightsquigarrow \lambda \mathtt{x}.e_t : G \to G_1}$$

$$\textsc{App}\ \frac{\Gamma \vdash_G e_{1s} \rightsquigarrow e_{1t} : G \qquad \Gamma \vdash_G e_{2s} \rightsquigarrow e_{2t} : G' \qquad dom\,(G) \sim G'}{\Gamma \vdash_G e_{1s}\ e_{2s} \rightsquigarrow (\llparenthesis dom\,(G) \to cod\,(G) \Leftarrow G \rrparenthesis e_{1t}\ \llparenthesis dom\,(G) \Leftarrow G' \rrparenthesis e_{2t}) : cod\,(G)}$$

$$\textsc{For}\ \frac{\Gamma \vdash_G e_{1s} \rightsquigarrow e_{1t} : G_1 \qquad ext_L(G_1) = G \qquad \Gamma, \mathtt{x} \mapsto G \vdash_G e_{2s} \rightsquigarrow e_{2t} : G_2 \qquad l\ fresh}{\Gamma \vdash_G \textbf{for}\ \mathtt{x}\ \textbf{in}\ e_{1s}\ \textbf{do}\ e_{2s} \rightsquigarrow \textbf{for}\ \mathtt{x}\ \textbf{in}\ \llparenthesis [G] \Leftarrow G_1 \rrparenthesis e_{1t}\ \textbf{do}\ e_{2t} : G_2}$$

$$\textsc{Let}\ \frac{\Gamma \vdash_G e_{1s} \rightsquigarrow e_{1t} : G_1 \qquad \Gamma, \mathtt{x} \mapsto G_1 \vdash_G e_{2s} \rightsquigarrow e_{2t} : G}{\Gamma \vdash_G \textbf{let}\ \mathtt{x} = e_{1s}\ \textbf{in}\ e_{2s} \rightsquigarrow \textbf{let}\ \mathtt{x} = e_{1t}\ \textbf{in}\ e_{2t} : G}$$

$$\textsc{Letrec}\ \frac{\Gamma, \mathtt{x} \mapsto G_1 \vdash_G e_{1s} \rightsquigarrow e_{1t} : G_1 \qquad \Gamma, \mathtt{x} \mapsto G_1 \vdash_G e_{2s} \rightsquigarrow e_{2t} : G}{\Gamma \vdash_G \textbf{letrec}\ \mathtt{x} = e_{1s}\ \textbf{in}\ e_{2s} \rightsquigarrow \textbf{letrec}\ \mathtt{x} = e_{1t}\ \textbf{in}\ e_{2t} : G}$$

$$\textsc{Ref}\ \frac{\Gamma \vdash_G e_s \rightsquigarrow e_t : G}{\Gamma \vdash_G \textbf{ref}\ e_s \rightsquigarrow \textbf{ref}\ e_t : \mathtt{ref}\ G} \qquad \textsc{Deref}\ \frac{\Gamma \vdash_G e_s \rightsquigarrow e_t : G_1 \qquad G = ext_R(G_1)}{\Gamma \vdash_G \mathtt{!}e_s \rightsquigarrow \mathtt{!}\llparenthesis \mathtt{ref}\ G \Leftarrow G_1 \rrparenthesis e_t : G}$$

$$\textsc{Assign}\ \frac{\Gamma \vdash_G e_{1s} \rightsquigarrow e_{1t} : G_1 \qquad G = ext_R(G_1) \qquad \Gamma \vdash_G e_{2s} \rightsquigarrow e_{2t} : G_2 \qquad G \sim G_2}{\Gamma \vdash_G e_{1s} \mathtt{:=} e_{2s} \rightsquigarrow \llparenthesis \mathtt{ref}\ G \Leftarrow G_1 \rrparenthesis e_{1t} \mathtt{:=} \llparenthesis G \Leftarrow G_2 \rrparenthesis e_{2t} : \mathtt{ref}\ G}$$

$$\llparenthesis G \Leftarrow G \rrparenthesis e_t = e_t \qquad \llparenthesis G_1 \Leftarrow G \rrparenthesis e_t = \lceil G_1 \Leftarrow G \rceil e_t$$

$$dom\,(G_1 \to G_2) = G_1 \qquad dom\,(\mathtt{Dyn}) = \mathtt{Dyn} \qquad cod\,(G_1 \to G_2) = G_2 \qquad cod\,(\mathtt{Dyn}) = \mathtt{Dyn}$$

$$ext_L([G]) = G \qquad ext_L(\mathtt{Dyn}) = \mathtt{Dyn} \qquad ext_R(\mathtt{ref}\ G) = G \qquad ext_R(\mathtt{Dyn}) = \mathtt{Dyn}$$

Fig. 5. Cast insertion rules.

being assigned to, and this similarly generates a cast. Additionally, the type of the expression being stored is cast to to the underlying type of the reference.

4.4 Cost of Wrapped Casts

The cast insertion rules in Figure 5 can insert casts involving function types. Since whether the cast will be successful or not cannot be checked at definition time, the usual *guarded* approach handles functions by dynamically creating function *proxies* that wrap underlying functions and cast their inputs and outputs when they are called [Siek and Taha 2006]. The costs of such casts thus depend on how the are used, which our cost model so far does not consider. To illustrate, consider the following expression.

$$\textbf{let}\ \mathtt{f} = \lceil \mathtt{Int} \to \mathtt{Int} \Leftarrow \mathtt{Dyn} \to \mathtt{Dyn} \rceil \lambda \mathtt{x}.\mathtt{x}\ \textbf{in}\ \mathtt{f}\ 1 + \mathtt{f}\ 2$$

Following the ideas in Section 4.2, we will assign a potential cost $\Lambda x.(0, x)$ to `f` (since there are no casts in the function body), and so the calls at `f 1` and `f 2` each generate a cost of $\Lambda x.(0, x)\ 0$, which reduces to 0. This, however, does not match the cost of guarded semantics where each call induces two casts, one from `Int` to `Dyn` and the other from `Dyn` to `Int`.

We want to adapt the potential assigned to `f` so that it includes the costs of casts in the generated proxies. However, the challenge is that to properly create these potentials, the cost analysis, which

$$\searrow(\lceil G_1 \to G_2 \Leftarrow \texttt{Dyn}\rceil e_t) = \searrow(\lceil G_1 \to G_2 \Leftarrow \texttt{Dyn} \to \texttt{Dyn}\rceil(\searrow e_t)) \quad (1)$$

$$\searrow(\lceil \texttt{Dyn} \Leftarrow G_1 \to G_2\rceil e_t) = \searrow(\lceil \texttt{Dyn} \to \texttt{Dyn} \Leftarrow G_1 \to G_2\rceil(\searrow e_t)) \quad (2)$$

$$\searrow(\lceil G_3 \to G_4 \Leftarrow G_1 \to G_2\rceil e_t) = \lambda \texttt{x}.(\searrow(\lceil G_4 \Leftarrow G_2\rceil(\searrow e_t)) \ \searrow(\lceil G_2 \Leftarrow G_3\rceil \texttt{x})) \quad (3)$$

$$\searrow(\textbf{ref}\ e_t) = \textbf{let}\ \texttt{y} = \searrow(e_t)\ \textbf{in}\ \lambda \texttt{x}.\texttt{y} \quad (4)$$

$$\searrow(\texttt{!}e_t) = (\searrow e_t)\ () \quad (5)$$

$$\searrow(e_{1t} \texttt{:=} e_{2t}) = \searrow(e_{1t})\ \searrow(e_{2t}) \quad (6)$$

$$\searrow(\lceil \texttt{ref}\ G_2 \Leftarrow \texttt{Dyn}\rceil e_t) = \searrow(\lceil \texttt{ref}\ G_2 \Leftarrow \texttt{ref Dyn}\rceil(\searrow e_t)) \quad (7)$$

$$\searrow(\lceil \texttt{Dyn} \Leftarrow \texttt{ref}\ G_2\rceil e_t) = \searrow(\lceil \texttt{ref Dyn} \Leftarrow \texttt{ref}\ G_2\rceil(\searrow e_t)) \quad (8)$$

$$\searrow(\lceil \texttt{ref}\ G_2 \Leftarrow \texttt{ref}\ G_1\rceil e_t) = \lambda \texttt{x}.(\searrow(\lceil G_2 \Leftarrow G_1\rceil(\searrow e_t)) \ \searrow(\lceil G_1 \Leftarrow G_2\rceil \texttt{x})) \quad (9)$$

$$\textit{otherwise}\ \searrow(\lceil G_2 \Leftarrow G_1\rceil e_t) = \lceil G_2 \Leftarrow G_1\rceil(\searrow e_t) \quad (10)$$

Fig. 6. Transformation rules after cast insertion. Proxies usually inserted at runtime after checking certain values are expanded syntactically, where possible. The *otherwise* in case (10) means that this rule applies when all others fail.

is static, need to know about the proxies, which are created dynamically. The trick is that we transform the program before analysis to syntactically include the proxies that will be generated at runtime. Specifically, we transform each function cast into a lambda expression that contains casts within its body, thereby creating a potential whose cost is not 0. The example above is transformed into the following expression.

$$\textbf{let}\ \texttt{f} = \lambda \texttt{y}.\lceil \texttt{Int} \Leftarrow \texttt{Dyn}\rceil(\lambda \texttt{x}.\texttt{x}\ \lceil \texttt{Dyn} \Leftarrow \texttt{Int}\rceil \texttt{y})\ \textbf{in}\ \texttt{f 1 + f 2}$$

Now `f` has a potential $\Lambda y.(b + c, y)$, and each application `f 1` and `f 2` will be assigned cost $b + c$, yielding a total cost of $2(b + c)$, which matches the expected behavior of the guarded semantics for higher-order casts.

Similarly, for casts involving reference types, the guarded semantics will create a proxy that induces casts on all future dereferences and assignments. For such casts, we reuse the trick described above of embedding lambdas representing the proxies into the program before analysis. Correspondingly, we transform dereferences and assignments into lambda applications. Interestingly, this idea can treat proxied references (those that require casts) and raw references (those that do not) uniformly. To illustrate, consider the following expression, where `y` has type `Dyn`.

$$\textbf{let}\ \texttt{g} = \lambda \texttt{x}.\texttt{!x} * \texttt{!}(\textbf{ref}\ 1)\ \textbf{in}\ \texttt{f}\ \lceil \texttt{ref Int} \Leftarrow \texttt{ref Dyn}\rceil(\textbf{ref}\ \texttt{y})$$

Note that this expression contains both a proxied reference $\lceil$`ref Int` $\Leftarrow$ `ref Dyn`$\rceil$(**ref** `y`) and a raw reference **ref** 1, which are transformed into λ`a.`$\lceil$`Int` $\Leftarrow$ `Dyn`$\rceil y$ and λ`d.`1, respectively (in reality, a let binding is used to evaluate the expression in the `ref` body before wrapping it in a lambda, but we just directly wrap the expressions here for simplicity). Each dereference is transformed into an application by applying it to a unit value, (). Overall, the transformation yields the following expression.

$$\textbf{let}\ \texttt{g} = \lambda \texttt{x}.(\texttt{x}\ () * (\lambda d.1)\ ())\ \textbf{in}\ \texttt{f}\ (\lambda \texttt{a}.\lceil \texttt{Int} \Leftarrow \texttt{Dyn}\rceil \texttt{y})$$

Now let us analyze the cast costs. For the proxied reference, the potential is $\Lambda a.(c, y)$, and for the raw reference, the potential is $\Lambda d.(0, 0)$. When they are dereferenced, the corresponding applications lead to the costs $(\Lambda a.(c, y))\ 0$ and $(\Lambda d.(0, 0))\ 0$, which are c and 0, respectively, matching the expected costs of guarded semantics on dereferences.

$$\textsc{Var}\ \frac{\mathsf{x} \mapsto P \in \Phi}{\Phi \vdash_c \mathsf{x} \mid (0, P)} \qquad \textsc{Abs}\ \frac{\Phi, \mathsf{x} \mapsto x \vdash_c e_t \mid A}{\Phi \vdash_c \lambda x.e_t \mid (0, \Lambda x.A)} \qquad \textsc{App}\ \frac{\Phi \vdash_c e_{1t} \mid (C_1, P_1) \qquad \Phi \vdash_c e_{2t} \mid (C_2, P_2)}{\Phi \vdash_c e_{1t}\ e_{2t} \mid C_1 + C_2 \oplus (P_1\ P_2)}$$

$$\textsc{For}\ \frac{\Phi \vdash_c e_{1t} \mid (C_1, P_1) \qquad \Phi, \mathsf{x} \mapsto P_1 \vdash_c e_{2t} \mid (C_2, P_2) \qquad l\ \mathit{fresh}}{\Phi \vdash_c \textbf{for}\ \mathsf{x}\ \textbf{in}\ e_{1t}\ \textbf{do}\ e_{2t} \mid (C_1 + l \cdot C_2, l \cdot P_2)}$$

$$\textsc{Let}\ \frac{\Phi \vdash_c e_{1t} \mid (C_1, P_1) \qquad \Phi, \mathsf{x} \mapsto P_1 \vdash_c e_{2t} \mid (C_2, P)}{\Phi \vdash_c \textbf{let}\ \mathsf{x} = e_{1t}\ \textbf{in}\ e_{2t} \mid (C_1 + C_2, P)}$$

$$\textsc{Letrec}\ \frac{\begin{array}{c}\Phi, \mathsf{x} \mapsto x \vdash_c e_{1t} \mid (C_1, P_1) \qquad S = \mathit{Apps}(x, P_1) \\ n = |S| \qquad P' = P_1 \ominus S \qquad l\ \mathit{fresh} \qquad \Phi, \mathsf{x} \mapsto (l \cdot n^l \cdot\ P') \vdash_c e_{2t} \mid (C_2, P_2)\end{array}}{\Phi \vdash_c \texttt{letrec}\ \mathsf{x} = e_{1t}\ \texttt{in}\ e_{2t} \mid (C_1 + C_2, P_2)}$$

$$\textsc{Cast}\ \frac{\Phi \vdash_c e_t \mid (C_1, P) \qquad \mathit{cost}(\lceil G_2 \Leftarrow G_1 \rceil) = C_2}{\Phi \vdash_c \lceil G_2 \Leftarrow G_1 \rceil e_t \mid (C_1 + C_2, P)}$$

Fig. 7. Cost semantics.

Overall, by transforming function casts and dereferences into lambda abstractions, we can reuse the idea of potentials to precisely estimate the costs of these casts. In Figure 6, we present rules for transforming expressions as described above, where $\searrow(e_t)$ applies the transformations to e_t. In the figure, we present rules that are relevant to casts only and ignore rules for other constructs of e_t, which recursively apply the transformation to their subterms, if applicable. The first three rules transform casts with function types. The next three transform reference expressions into lambdas and applications. The next three handle casts with references. The final rule terminates the recursive transformation in rules 3 and 9 when they arrive at casts between base types.

4.5 Cost Computing Rules

This subsection defines a cost semantics that computes a cost approximation for any expression transformed by $\searrow$. The cost semantics is presented in Figure 7. The rules have the general form, $\Phi \vdash_c e_t \mid A$, meaning that expression e_t has approximation A in the context of cost environment Φ.

In rule Var, a variable reference x has no immediate cost since the language is call-by-value, but may have a potential cost that is retrieved from the cost environment. For example, a variable f can reference a function, which would have an abstraction for its potential. The cost of abstractions in rule Abs is 0 since they cause no evaluation, but their potential is an abstraction of the form $\Lambda x.A$.

In rule App, the potential of an application is the application of the corresponding potentials. The cost of an application is the cost of the two subexpressions, plus the cost of these two casts, and the cost of the potential application. The term $C_1 + C_2 \oplus (P_1\ P_2)$ is used to pairwise add the cost of the potential application after it evaluates. For example:

$$1 \oplus (\Lambda x.(1, x)\ 0) = 1 \oplus [0/x](1, x) = (1 + 1, 0)$$

This term is sometimes left unevaluated. For example, the approximation for a function with a higher-order argument, such as $\lambda \mathsf{x}.\mathsf{x}\ 1$, is $\Lambda x.(0 \oplus (x\ 0))$, which will evaluate after we substitute a corresponding potential of the higher-order argument in a function call.

In rule For, a fresh loop label l is introduced to represent the number of loop iterations. The cost of the loop is then the cost of evaluating e_1, plus l times the cost of evaluating e_2. The potential is constructed similarly. For let expressions in rule Let, the cost and potential is computed similarly to For except that the body is evaluated only once.

The rule Letrec assigns costs to recursive expressions and bindings. Since e_{1t} refers to x, the potential P_1 for e_{1t} must contain potential applications that apply x to some other potentials. Moreover, these applications are connected by $\oplus$. We use $Apps(x, P_1)$ to collect all such potential applications into S. We then use n to measure the cardinality of S. For example, the value of n in a naive recursive definition of a function to compute the Fibonacci sequence would be 2, since it involves two recursive function calls. We also use the operation $\ominus$ to remove all the applications in S from P_1 and assign the result to P'. As a result, P' contains no further potential applications applying x to some term. The recursion is then estimated to have the cost $l * n^l$, where l is a fresh label estimating the size of the input. We then use this cost to compute the cost for e_{2t}, the overall cost for the whole construct. In general, static cost analysis for recursive programs is difficult and an area receiving significant recent research [Danner et al. 2015; Hoffmann et al. 2017]. This difficulty is further exacerbated in our case since the type information that is available in other static cost analyses is unavailable in the gradual type setting. Our costs for recursions are a coarse upper bound, and we assume the input to recursive calls strictly decrease in size, similar to previous cost analyses [Danner et al. 2015]. Nevertheless, this cost estimation works quite well in practice because the bounds for recursion do not affect relative cost comparisons, in particular the bounds are shared for nearby type configurations in the cost lattice.

Finally, the rule Cast assigns a cost to each expression being cast, whose cost is that of the underlying expression plus that of the cast, according to the *cost* function from Section 4.1. Rules for approximating the costs of references and higher-order casts are handled by costs for abstractions and applications after using the transformation procedure in Figure 6.

4.6 Properties

In the following lemmas and theorems, we use the judgment form $\Phi; \Gamma \vdash_{GC} e_s \rightsquigarrow e_t : G \mid A$, which is equivalent to the judgment $\Gamma \vdash_G e_s \rightsquigarrow e_t : G$ followed by $\Phi \vdash_c \searrow (e_t) \mid A$. Essentially, the judgment can be read as: under Γ and Φ, e_s has type G, is translated to e_t, and has the cost approximation A.

Before we present the most important properties of our cost semantics, we present some simple lemmas relating terms in the source language to terms in the cost language. The first lemma states that a bound variable, if referenced, affects the potential of its abstraction.

Lemma 4.1. *If* $\Phi; \Gamma \vdash_{GC} \lambda \mathrm{x}.e_s \rightsquigarrow \lambda \mathrm{x}.e_t : G \mid (C, P)$ *and* $\mathrm{x} \in vars(e_s)$*, then* $x \in vars(P)$.

This lemma confirms that the potential costs of an argument, which may be a function with its own costs, are reflected in the cost approximation of the function that uses it.

The second lemma states that substitution in the source language corresponds to substitution in the cost language.

Lemma 4.2. *Let* $\Phi; \Gamma \vdash_{GC} e_s \rightsquigarrow e_t : G \mid (C, P)$ *where* $\mathrm{x} \in vars(e_s)$, $\mathrm{x} \mapsto x \in \Phi$, *and* $\mathrm{x} : G'$.
If $\Phi; \Gamma \vdash_{GC} e_s' \rightsquigarrow e_t' : G' \mid (C', P')$*, then the potential for* $[e_s'/\mathrm{x}]e_s$ *is* $[P'/x]P$.

This establishes how potential applications can insert the overhead of casts in the argument terms into the body of a potential abstraction. Together, these two lemmas establish the correspondence of function abstraction and application at the source level and the cost level. Moreover, since we also reason about the overhead of using references by translating them into lambdas and then applying our cost semantics, it is imperative that abstractions and applications be modeled correctly.

Target expr.	e_t^v	::=	$\ldots \mid \lceil M \Leftarrow M \rceil e_t^v \mid d\langle \lceil M \Leftarrow M \rceil, \lceil M \Leftarrow M \rceil \rangle e_t^v$
Costs	C^v	::=	$\ldots \mid d\langle C^v, C^v \rangle$
Potentials	P^v	::=	$\ldots \mid d\langle P^v, P^v \rangle$
Approximations	A^v	::=	$\ldots \mid (C^v, P^v) \mid C^v \oplus P^v$
Variational types	V	::=	$\gamma \mid V \to V \mid [V] \mid \texttt{ref}\ V \mid d\langle V, V \rangle$
Migrational types	M	::=	$\gamma \mid M \to M \mid [M] \mid \texttt{Dyn} \mid \texttt{ref}\ M \mid d\langle M, M \rangle$
Configuration	K	::=	$\varnothing \mid K, \mathsf{x} \mapsto G$
Choice env.	Ω	::=	$\varnothing \mid \Omega, \mathsf{x} \mapsto M$

Fig. 8. Syntax for variational cost analysis. Definitions of variational artifacts (e.g. e_t^v) extend the syntax for non-variational counterparts (e.g. e_t) in Figure 4.

We now establish the most important properties of our cost semantics. The following theorem states that our cost semantics terminates, provided that the program after translation (e_t) is terminating.

Theorem 4.3. *[Cost Derivation Termination] For any terminating program e_s, $\Phi; \Gamma \vdash_{GC} e_s \rightsquigarrow e_t : G \mid A$ terminates and produces the approximation A.*

Finally, the most important result is that our cost semantics bounds the number of casts of the program, provided that its recursions, if any, are applied to smaller arguments.

Theorem 4.4. *[Costs are Upper Bounds] If $\Phi; \Gamma \vdash_{GC} e_s \rightsquigarrow e_t : G \mid (C, P)$, then C is greater than or equal to the number of casts performed when executing e_t after replacing loop labels by the number of iterations performed.*

The proofs[3] of these two theorems are produced by induction over the rules in Figures 5, referencing the rules in Figure 7. The proofs also relate the dynamic semantics for the language, which is a fairly standard semantics with a few extensions for loops and other constructs.

5 COSTS FOR ALL CONFIGURATIONS

In this section we extend the formalization in Section 4 to make it variational, enabling us to efficiently estimate costs for all possible type configurations of a program. Thus, instead of assigning a gradual type and a cost to each program, the rules in this section assign a variational gradual type, called a *migrational type* [Campora et al. 2018] and a variational cost to each program. The migrational type of a program represents the type of all possible type configurations, while its variational cost encodes the cost lattice of migrating to each configuration, as illustrated in Figure 2.

5.1 Syntax

In Figure 8, we extend the syntax defined in Figure 4 to accommodate the variational analysis. In particular, we extend costs and potentials with a choice construct, as described in Section 2.2. This enables the representation of variational costs and variational potentials. The addition of choices to types yields two new domains, *variational types* (static types with choices) and *migrational types* (gradual types with choices), ranged over by V and M, respectively. Casts in the target language are made variational by allowing casting to a migrational type.

Figure 8 also defines *configurations*, ranged over by K, and *choice environments*, ranged over by Ω. A configuration is a mapping from initially dynamic parameters to gradual types, denoting

[3]The proofs are given in the longer version of this paper at http://www.ucs.louisiana.edu/~sxc2311/ws/techreport/long-mp.pdf.

their type assignments for a particular configuration. A choice environment is a mapping from parameters to migrational types and keeps track of the type assignments used to generate the full configuration space for a given program. For example, if a program has dynamic parameters x and y, then the choice environment $\Omega = \{x \mapsto B\langle \texttt{Dyn}, \texttt{Int}\rangle, y \mapsto D\langle \texttt{Dyn}, \texttt{Int}\rangle\}$ encodes four configurations (each of x and y can be Dyn or Int). For simplicity, we assume that all parameter names in the program are all unique. Configurations and choice environments are used to establish the correctness of our variational cost analysis by making explicit the relation between the rules in this section and those in Section 4.3.

Although migrational types now appear in casts in the target language, we do not extend its dynamic semantics since in the implementation programs with variational casts are not executed, only analyzed. After the analysis, the programmer would select a particular type configuration by applying a complete decision to the program that eliminates migrational casts and yields a runnable program.

5.2 Variational Cost Analysis

The cost semantics defined in Section 4 requires some changes to accommodate casts to migrational types. First, we extend the cost function with the following new rules, which essentially maps the cost calculation over cast choices, preserving the results in cost choices.

$$cost(\llbracket d\langle M_1, M_2\rangle \Leftarrow M\rrbracket) = d\langle cost(\llbracket M_1 \Leftarrow M\rrbracket), cost(\llbracket M_2 \Leftarrow M\rrbracket)\rangle$$
$$cost(\llbracket M \Leftarrow d\langle M_1, M_2\rangle\rrbracket) = d\langle cost(\llbracket M \Leftarrow M_1\rrbracket), cost(\llbracket M \Leftarrow M_2\rrbracket)\rangle$$

Next, we similarly extend the transforming procedure ($\searrow$) to push the translation into choices. Following the same idea, we extend the cost relation $\vdash_c$ in Figure 7 to deal with variations. We name these extended relations $\searrow^v$ and $\vdash_c^v$, respectively. A interesting bit in $\vdash_c^v$ is that our cost function now yields variational costs, and so we must define how arithmetic works on such values. Intuitively, any basic operation on a variational cost can be performed by pushing the operations into the choices. For example, $c + B\langle 2c, 3c\rangle = B\langle c+2c, c+3c\rangle = B\langle 3c, 4c\rangle$. We omit a formal definition here since it is straightforward.

In Figure 9, we present the revised set of type-directed cast insertion rules to support migrational types. The judgment has the form $\Gamma \vdash^v e_s \rightsquigarrow e_t^v : M \mid \Omega$, which states that under type environment Γ, the source expression e_s has type M and is translated to the target expression e_t^v after casts are inserted, where type change information is recorded in Ω. The cast insertion rules for variables, loops, let expressions, and references are nearly identical to the corresponding rules in Figure 5, except that they now use the extended syntax.

There are now two rules for abstractions, one for statically typed parameters (AbsV) and the other for dynamically typed parameters (AbsDynV). The AbsDynV rule is where variation is injected into the target program so that it captures the whole migration space. The choice type in AbsDynV represents the fact that there are two possibilities during program migration: leaving the parameter dynamically typed, or changing it to a static type. Correspondingly, when we carry out our cost analysis there will be two different costs for the body. For example, in λx.x + 1, the reference to x in the body must be cast using $\lceil \texttt{Int} \Leftarrow d\langle \texttt{Dyn}, \texttt{Int}\rangle\rceil$, which will be assigned the cost $d\langle c, 0\rangle$ since we incur a cast when x has type Dyn and no cast when it has type Int. In AbsDynV, we also extend Ω to record that we assigned the type $d\langle \texttt{Dyn}, V\rangle$ to the parameter.

Variation complicates the treatment of function applications. First, the *dom* and *cod* functions are extended to support migrational types. Second, the type consistency relation used in the original App rule of Figure 5 is replaced by the *compatibility* relation ($\approx$) defined in [Campora et al. 2018], which extends type consistency to also support variational type equivalence (see Section 2.2). Two

$$\textsc{VarV}\ \frac{\mathsf{x} \mapsto M \in \Gamma}{\Gamma \vdash^{v} \mathsf{x} \rightsquigarrow \mathsf{x} : M \mid \Omega} \qquad \textsc{AbsV}\ \frac{\Gamma, \mathsf{x} \mapsto T \vdash^{v} e_s \rightsquigarrow e_t^{v} : M \mid \Omega}{\Gamma \vdash^{v} \lambda \mathsf{x} : T.e_s \rightsquigarrow \lambda \mathsf{x}.e_t^{v} : T \to M \mid \Omega}$$

$$\textsc{AbsDynV}\ \frac{\Gamma, \mathsf{x} \mapsto d\langle \mathtt{Dyn}, V \rangle \vdash^{v} e_s \rightsquigarrow e_t^{v} : M \mid \Omega \qquad d\ \mathit{fresh}}{\Gamma \vdash^{v} \lambda \mathsf{x}.e_s \rightsquigarrow \lambda x.e_t^{v} : d\langle \mathtt{Dyn}, V \rangle \to M \mid \Omega \cup \{x \mapsto d\langle \mathtt{Dyn}, V \rangle\}}$$

$$\textsc{AppV}\ \frac{\Gamma \vdash^{v} e_{1s} \rightsquigarrow e_{1t}^{v} : M_1 \mid \Omega_1 \qquad \Gamma \vdash^{v} e_{2s} \rightsquigarrow e_{2t}^{v} : M_2 \mid \Omega_2 \qquad \mathit{dom}(M_1) \approx M_2}{\Gamma \vdash^{v} e_{1s}\ e_{2s} \rightsquigarrow \llbracket \mathit{dom}(M_1) \to \mathit{cod}(M_1) \Leftarrow M_1 \rrbracket e_{1t}^{v}\ \llbracket \mathit{dom}(M_1) \Leftarrow M_2 \rrbracket e_{2t}^{v} : \mathit{cod}(M_1) \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{ForV}\ \frac{\Gamma \vdash^{v} e_{1s} \rightsquigarrow e_{1t}^{v} : M_1 \mid \Omega_1 \qquad \mathit{ext}_L(M_1) = M \qquad \Gamma, \mathsf{x} \mapsto M \vdash^{v} e_{2s} \rightsquigarrow e_{2t}^{v} : M_2 \mid \Omega_2}{\Gamma \vdash^{v} \mathbf{for}\ \mathsf{x}\ \mathbf{in}\ e_{1s}\ \mathbf{do}\ e_{2s} \rightsquigarrow \mathbf{for}\ \mathsf{x}\ \mathbf{in}\ \llbracket [M] \Leftarrow M_1 \rrbracket e_{1t}^{v}\ \mathbf{do}\ e_{2t}^{v} : M_2 \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{LetV}\ \frac{\Gamma \vdash^{v} e_{1s} \rightsquigarrow e_{1t}^{v} : M_1 \mid \Omega_1 \qquad \Gamma, \mathsf{x} \mapsto M_1 \vdash^{v} e_{2s} \rightsquigarrow e_{2t}^{v} : M \mid \Omega_2}{\Gamma \vdash^{v} \mathbf{let}\ \mathsf{x} = e_{1s}\ \mathbf{in}\ e_{2s} \rightsquigarrow \mathbf{let}\ \mathsf{x} = e_{1t}^{v}\ \mathbf{in}\ e_{2t}^{v} : M \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{LetrecV}\ \frac{\Gamma, \mathsf{x} \mapsto M_1 \vdash^{v} e_{1s} \rightsquigarrow e_{1t}^{v} : M_1 \mid \Omega_1 \qquad \Gamma, \mathsf{x} \mapsto M_1 \vdash^{v} e_{2s} \rightsquigarrow e_{2t}^{v} : M \mid \Omega_2}{\Gamma \vdash^{v} \mathbf{letrec}\ \mathsf{x} = e_{1s}\ \mathbf{in}\ e_{2s} \rightsquigarrow \mathbf{letrec}\ \mathsf{x} = e_{1t}^{v}\ \mathbf{in}\ e_{2t}^{v} : M \mid \Omega_1 \cup \Omega_2}$$

$$\textsc{RefV}\ \frac{\Gamma \vdash^{v} e_s \rightsquigarrow e_t^{v} : M \mid \Omega}{\Gamma \vdash^{v} \mathbf{ref}\ e_s \rightsquigarrow \mathbf{ref}\ e_t^{v} : \mathtt{ref}\ M \mid \Omega} \qquad \textsc{DerefV}\ \frac{\Gamma \vdash^{v} e_s \rightsquigarrow e_t^{v} : M_1 \mid \Omega \qquad M = \mathit{ext}_R(M_1)}{\Gamma \vdash^{v} !e_s \rightsquigarrow !\llbracket \mathtt{ref}\ M \Leftarrow M_1 \rrbracket e_t^{v} : M \mid \Omega}$$

$$\textsc{AssignV}\ \frac{\begin{array}{c}\Gamma \vdash^{v} e_{1s} \rightsquigarrow e_{1t}^{v} : M_1 \mid \Omega_1 \\ M = \mathit{ext}_R(M_1) \qquad \Gamma \vdash^{v} e_{2s} \rightsquigarrow e_{2t}^{v} : M_2 \mid \Omega_2 \qquad M \approx M_2\end{array}}{\Gamma \vdash^{v} e_{1s} := e_{2s} \rightsquigarrow \llbracket \mathtt{ref}\ M \Leftarrow M_1 \rrbracket e_{1t}^{v} := \llbracket M \Leftarrow M_2 \rrbracket e_{2t}^{v} : \mathtt{ref}\ M \mid \Omega_1 \cup \Omega_2}$$

$$\begin{array}{lll}
\mathit{dom}(M_1 \to M_2) = M_1 & \mathit{dom}(\mathtt{Dyn}) = \mathtt{Dyn} & \mathit{dom}(d\langle M_1, M_2 \rangle) = d\langle \mathit{dom}(M_1), \mathit{dom}(M_2) \rangle \\
\mathit{cod}(M_1 \to M_2) = M_2 & \mathit{cod}(\mathtt{Dyn}) = \mathtt{Dyn} & \mathit{cod}(d\langle M_1, M_2 \rangle) = d\langle \mathit{cod}(M_1), \mathit{cod}(M_2) \rangle \\
\mathit{ext}_L([M]) = M & \mathit{ext}_L(\mathtt{Dyn}) = \mathtt{Dyn} & \mathit{ext}_L(d\langle M_1, M_2 \rangle) = d\langle \mathit{ext}_L(M_1), \mathit{ext}_L(M_2) \rangle \\
\mathit{ext}_R(\mathtt{ref}\ M) = M & \mathit{ext}_R(\mathtt{Dyn}) = \mathtt{Dyn} & \mathit{ext}_R(d\langle M_1, M_2 \rangle) = d\langle \mathit{ext}_R(M_1), \mathit{ext}_R(M_2) \rangle
\end{array}$$

Fig. 9. Cast insertion rules after adding variational types to our system. The operations *dom*, *cod*, ext_L, and ext_R are undefined for cases that are not listed here.

types M_1 and M_2 are compatible, written as $M_1 \approx M_2$, if they are consistent or compatible under any selection. For example, $B\langle \mathtt{Int}, \mathtt{Dyn} \rangle \approx B\langle \mathtt{Dyn}, D\langle \mathtt{Bool}, \mathtt{Int} \rangle \rangle$, since selecting $B.1$ in both types yields $\mathtt{Int} \sim \mathtt{Dyn}$, while selecting $B.2$ in both yields $\mathtt{Dyn} \approx D\langle \mathtt{Bool}, \mathtt{Int} \rangle$.

5.3 Properties

In this subsection, we prove the correctness of our variational cost analysis by showing that it is equivalent to generating all type configurations and applying the cost analysis from Section 4 to each one individually. We introduce a new judgment $\Phi; \Gamma \vdash^{v} e_s \rightsquigarrow e_t^{v} : M \mid A^{v} \mid \Omega$ as shorthand for $\Gamma \vdash^{v} e_s \rightsquigarrow e_t^{v} : M \mid \Omega$ followed by $\Phi \vdash_c^{v} \searrow^{v} (e_t^{v}) \mid A^{v}$.

We first extend selection, defined in Section 2.2, to choice environments by applying selections to its range. For example, let $\Omega_a = \{\mathsf{x} \mapsto B\langle \mathtt{Dyn}, \mathtt{Int} \rangle, \mathsf{y} \mapsto D\langle \mathtt{Dyn}, \mathtt{Int} \rangle\}$, then $\lfloor \Omega_a \rfloor_{\{B.1, D.2\}} = K_a$, where $K_a = \{\mathsf{x} \mapsto \mathtt{Dyn}, \mathsf{y} \mapsto \mathtt{Int}\}$. We also need a way to apply a configuration to a typing process in order

to conveniently type different individual type configurations. We write $\Phi; K\Gamma \vdash_{GC} e_{1s} \rightsquigarrow e_{1t} : G \mid A$ to express that the cost analysis (Section 4.6) is directed by the configuration K. The configuration K overrides the type assignments in the environment Γ, so for each variable reference x, if $\mathtt{x} \mapsto G \in K$, then x has type G, otherwise it retrieves the type from Γ. Since we assume all parameters have unique names, this can be achieved without ambiguity. For example, let add = λx : Dyn.λy : Dyn.x+y and $\Phi; \Gamma \vdash_{GC} \mathtt{add} \rightsquigarrow e_{1t} : \mathtt{Int} \mid (2c, P_1)$, then e_{1t} includes two casts $\lceil \mathtt{Int} \Leftarrow \mathtt{Dyn} \rceil$ to the parameters x and y, whereas in $\Phi; K_a\Gamma \vdash_{GC} \mathtt{add} \rightsquigarrow e_{2t} : \mathtt{Int} \mid (c, P_2)$, then e_{2t} includes a cast $\lceil \mathtt{Int} \Leftarrow \mathtt{Dyn} \rceil$ to the parameter x only. The reason is that K_a forces the parameter y to have type Int.

We can now state the correctness of the rules in Figure 9 in two steps. First, Theorem 5.1 states that the cost of any configuration can be obtained through some variational cost calculation.

Theorem 5.1 (Variational Cost Completeness). *For any K, if $\Phi; K\Gamma \vdash_{GC} e_s \rightsquigarrow e_t : G \mid A$, then there exists some variational cost analysis $\Phi; \Gamma \vdash^{v} e_s \rightsquigarrow e_t^v : M \mid A^v \mid \Omega$ such that $e_t = \lfloor e_t^v \rfloor_\delta$, $A = \lfloor A^v \rfloor_\delta$, $G = \lfloor M \rfloor_\delta$, and $K = \lfloor \Omega \rfloor_\delta$, where δ can be decided by e_s and K.*

Given an expression e and a configuration K for e, the corresponding decision δ can be deduced by considering which alternative of each choice in e must be selected. For example, in the add example above, the decision for K_a is $\{B.1, D.2\}$.

According to Theorem 5.1, it is possible that we need to use different variational cost analyses to obtain costs for different configurations. Fortunately, the following theorem shows that we can find appropriate variational types for dynamic parameters such that the costs for all configurations can be obtained through just one variational cost analysis. We capture this idea in the following theorem.

Theorem 5.2 (Variational Costs Soundness and Efficiency). *Given any e_s for which the entire configuration space is well-typed, there exists some Ω for $\Phi; \Gamma \vdash^{v} e_s \rightsquigarrow e_t^v : M \mid A^v \mid \Omega$ such that $\Phi; \lfloor \Omega \rfloor_\delta \Gamma \vdash_{GC} e_s \rightsquigarrow \lfloor e_t^v \rfloor_\delta : \lfloor M \rfloor_\delta \mid \lfloor A^v \rfloor_\delta$ for any δ.*

The proof of both theorems follows by induction over the rules in Figure 9, connecting to the rules in Figure 5. The Ω in Theorem 5.2 can be found by migrational type inference [Campora et al. 2018]. Moreover, we can lift the restriction about the configuration space being well-typed by employing the pattern-constrained judgments introduced in that work, and our implementation uses this approach. In fact, the inference process and variational cost analysis can be combined and computed together, and we refer to this combined process as *Cost Space Typing* (CST).

5.4 Uses of Variational Costs

In Section 1.2, we outlined several scenarios programmers might encounter when migrating gradually typed programs. This subsection will briefly describe how, given a source program e_s, the CST $\Phi; \Gamma \vdash^{v} e_s \rightsquigarrow e_t^v : M \mid A^v \mid \Omega$ can be used to solve the problems posed by these scenarios.

Supporting the scenarios often involves comparing costs that refer to different loop variables and so are not comparable in principle. As a pragmatic solution, we simply instantiate all loop variables by the same large integer and then directly compare the resulting values. For example, given $2l_1 + 4$ and $3l_2 + 3$, we can instantiate both l_1 and l_2 by 100 and conclude that the first cost is cheaper than the second since $204 < 303$. The evaluation in Section 6 shows that this simple solution yields good results, although future work could attempt to constrain loop variables more precisely through program analysis.

In scenario S1, the goal is to maximize static safety first, then optimize for performance. Given the migrational typing, we can apply the method described by Campora et al. [2018] to extract the set of decisions corresponding to the most static migrations. Each decision δ_i in this set characterizes a

maximal set of parameters to be annotated in e_s. Finally, we select the δ_i that minimizes the value of $\lfloor A^{\upsilon} \rfloor_{\delta_i}$ after instantiating loop variables in the approximation A^{υ} as described above.

In scenario S2, the goal is to maximize performance, before maximizing the presence of static types for safety. To do this, we instantiate the loop variables then search A^{υ} for the decisions with minimum cost. The search procedure is a straightforward recursive function that keeps track of the lowest cost encountered as it builds up the decision(s) corresponding to the lowest costs encountered. We can allow the user to require certain functions or parameters to be annotated by simply selecting A^{υ} with the corresponding selectors before searching. This allows programmers to optimize performance while still enforcing certain typing goals. To find incremental migrations where at most c out of n parameters are migrated, we first compute the set of $\binom{n}{c}$ decisions where c parameters are annotated, then minimize A^{υ} under this set of decisions.

In scenario S3, the goal is to increase type-based safety guarantees without hurting performance. This can be achieved by searching A^{υ} for all costs that are lower than the current program. We can efficiently represent the result of such a search as a mask that can be applied to the typing results, similar to the treatment of type errors in Chen et al. [2012]. This would effectively produce a set of migrations that increase performance, after which we could maximize the static checking done in the remaining migrations, as described in the solution to S1.

Finally, in scenario S4, the goal is to understand why a particular migration exhibits degraded performance and to make informed decisions about the performance of different migrations. Supporting this scenario requires selecting the relevant migrations from A^{υ}, then translating the symbolic costs in each into explanations. For example, when considering configurations ① and ② in the program in Figure 1, we can report that ② has worse performance than ① since it inserts casts in the l_1 and l_2 loops in `part_A_times_u` and `part_At_times_u` that were not there before. Currently, Herder contains a basic implementation capable of reporting the differences in the set of loop labels between two configurations based on their costs.

6 IMPLEMENTATION AND EVALUATION

In this section we discuss Herder, a tool for carrying out variational cost analysis for Reticulated Python programs. In Section 6.1 we give an overview of its implementation. We then evaluate how well Herder realizes the capabilities described in Section 1.3, which are needed to support the scenarios described in Section 1.2. Since capability C1, migrational type inference, is provided by earlier work [Campora et al. 2018], we focus on C2 and C3. Specifically, in Sections 6.2 and 6.3, we evaluate how effectively and efficiently, respectively, Herder identifies performant configurations.

6.1 Implementation

Herder is implemented on top of Reticulated's guarded semantics [Vitousek et al. 2014], extending it with migrational type inference [Campora et al. 2018], cost analysis of casts, and variations. In addition to the constructs from Figure 4, our implementation supports conditionals and other Python constructs. We use the sympy package—which supports creating, algebraically manipulating, and substituting symbolic variables—to implement the approximation language.

6.2 Evaluation of Effectiveness

We evaluate the effectiveness of Herder by showing how often it can pick the most performant configurations among all valid configurations. Our sample programs are drawn from the Python benchmarks suite,[4] and are largely a subset of those used by Vitousek et al. [2017] to evaluate Reticulated, plus some programs from the scimark benchmark. We also generated three synthetic

[4]http://pyperformance.readthedocs.io/benchmarks.html

						Dynamic		Worst		Best		
Bench	LOC	#P	Analysis	Rec	Run	Time	Ratio	Time	Ratio	Time	Ratio	Top 3
float	64	6	1.67	3	69.7	103.6	1.49	128.7	1.83	61.5	0.88	✓
meteor	254	27	12.92	2	27.0	19.0	0.70	134.2	4.96	19.0	0.70	✓
nbody	157	16	3.34	6	64.3	65.6	1.02	174.0	2.70	59.6	0.93	✓
pidigits	68	5	0.71	4	8.1	47.0	5.83	45.7	5.65	8.0	0.99	✓
raytrace	448	66	63.60	47	30.1	56.7	1.88	109.3	3.63	30.1	1	✓
sm(FFT)	140	14	1.95	8	34.4	35.0	1.02	48.0	1.40	29.1	0.85	✗
sm(MC)	97	5	0.96	4	40.8	7.4	0.18	48.9	1.2	7.4	0.18	✓
sm(SOR)	110	16	7.19	16	37.8	97.9	2.59	120.7	3.19	37.8	1	✓
spectral	85	4	0.82	2	17.9	48.8	2.73	93.8	5.24	17.9	1	✓
syn_1	363	41	139.49	34	77.0	128.6	1.67	270.9	3.51	77.0	1	✓
syn_2	3710	787	1438.47	716	10.0	30.8	3.08	53.6	5.36	10.0	1	✓
syn_3	15213	2505	14607.84	2444	139.8	131.3	0.94	452.9	3.24	131.3	0.94	✓

Fig. 10. Evaluation of Herder. All timing results in the table are in seconds. The first column group provides basic stats about each benchmark: its name, lines of code, and the number of parameters it contains. The second group describes the results: the time to perform the analysis, the number of recommended static annotations, and the runtime of the resulting program. The third group compares the runtime of the resulting program to other potential configurations: the fully dynamic program, and the worst and best configurations identified identified in our sample. The final column indicates whether the configuration recommended by Herder is among the top 3 of most performant configurations (✓) or not (✗).

benchmarks (syn) to stress-test the performance of Herder, when the cost analysis contains deeply nested choices, a large number of choices, or a large number of code lines, respectively.

Among the 15 programs we evaluated, 3 of them, namely fankuch, nqueens, and pyflate, have very uniform performance, meaning that the addition of type annotations has very little effect on performance. As expected, Herder correctly identifies the most static configuration as having the best performance. For this reason, in the rest of this section, we will not discuss them in detail.

The metrics and evaluation results for the remaining 12 programs, are given in Figure 10. The first column group lists the name of each benchmark, its lines of code, and the total number of function parameters it contains. The number of parameters describes the size of the search space for our analysis. Since each parameter can either be statically annotated by the inferred type or not, a program with p parameters contains up to 2^p configurations.

The second column group reports the results of running Herder on each of the benchmarks. The Analysis column reports the time to perform the analysis in seconds. The Rec column reports the number of static type type annotations that Herder recommended be added to the program. Finally, the Run column reports the time in seconds needed to execute the target program after adding the recommended type annotations.

To evaluate the effectiveness of Herder at recommending performant configurations, we must compare the recommended configuration to other possibilities. For each benchmark, if there are fewer than 100 potential configurations, we generate and time every one. If there are more than 100 potential configurations (i.e. if the number of parameters is greater than 6), we randomly sample a set of 100 configurations, generating and timing the variant program for each. As a baseline, we also measure the runtime of the fully dynamically typed version of each benchmark.

The third column group reports results from this comparison. The first pair of columns reports the runtime of the fully dynamic version of the benchmark and the ratio of the dynamic version over our recommended configuration. The next two pairs of columns report the runtime and corresponding

ratios for the worst and best configuration in our sample for each benchmark. Finally, in the last column of the table, we report how the runtime of our recommended configuration ranked amongst the runtimes of all variants in the sample.

The results demonstrate that Herder is effective at finding performant configurations. In 11/12 cases, it recommends one of the top three configurations. When the recommendation does not achieve the best performance, its recommendation is usually within 15% of the optimal configuration. Moreover, we observe that the worst configuration is often 3-5x slower than Herder's pick. Some of the ratio results along with the top 3 result might seem odd. For example, sm(MC) has a poor "Best" ratio, yet Herder's pick remains in top 3, while sm(FFT) has a better "Best" ratio, but the pick is not in top 3. The reason for the poor "Best" ratio for sm(MC) is that the performance for the dynamic configuration was an outlier and Herder's pick had the fastest time compared to the rest. With respect to sm(FFT), most of the configurations had similar times, so the pick happened to fall out of top 3, even though the difference between the third best time and the pick's time was small.

Three interesting cases are the scimark Monte Carlo benchmark, sm(MC); the meteor_contest benchmark; and the syn_3 benchmarks, where the fully dynamic version of the program is faster than any gradually typed version. Currently, Herder only introduces choices to reason about alternative function parameter types but it infers return types directly. This means it does not consider the fully dynamic version of the program as a potential configuration. This limitation can be easily remedied by extending Herder to reason variationally about return types in addition to parameter types, allowing return types to remain dynamic when needed, so that accurate costs can be provided for these benchmarks.

Notice that the fully dynamic configuration does not necessarily have near-best performance, as is typical in Racket [Takikawa et al. 2016]. This is due to the fact that Python programs frequently unpack parameters via tuple assignment, which causes casts to be inserted. Typed literals can also cause some overhead in programs. Consequently, in the Rec column, we observe a fair amount of variability in whether the best configurations are more or less static. In some benchmarks, such as nbody, the recommended configuration has fewer type annotations, and in sm(MC), we know the best configuration contains no annotations. On the other hand, the best configuration for the scimark successive over-relaxation benchmark, sm(SOR), adds annotations to all parameters.

Effectiveness of supporting program migration scenarios We next evaluate how well Herder can support each migration scenario we outlined in Section 1.2. For scenarios S1 and S2 we use Figure 10 as evidence. Specifically, Figure 10 shows how well Herder can find globally performant configurations via the cost analysis, and consequently it directly supports S2.

To support S1, we first need to identify the migration space that contains all different migrations that maximize static type safety, which can be realized efficiently through the method described in Campora et al. [2018]. Performance optimization in this scenario then amounts to locating the most performant migration within the identified migration space, which is much smaller than the global space Herder searches for in Figure 10. Therefore, we believe that the effectiveness demonstrated in Figure 10 carries over to this scenario. As evidence of this argument, in our evaluation for S3 in the next paragraph, Herder similarly has to search a small space, and it achieved good results.

In scenario S3, we are considering the case where a recently added type annotation hurt performance, and asking Herder whether it is preferable to remove the annotation or add more annotations to restore performance. To evaluate this scenario, from the 9 non-synthetic programs in Figure 10, we generated 45 configurations by randomly adding type annotations. For 15 of these configurations, the performance is better than their corresponding untyped configurations. For the remaining 30 configurations, we asked Herder whether adding more type annotations (and if

Params	Herder	Brute-force	Dynamic
2	0.07	0.27	0.03
4	0.12	1.48	0.05
6	0.18	7.47	0.09
8	0.27	36.97	0.12
10	0.38	128.75	0.14
12	0.531	2545.33	0.17
51	13.03	-	4.12
99	40.41	-	13.09
195	181.86	-	46.76
387	675.64	-	178.94
807	2896.76	-	773.20

LOC	Herder	Brute-force	Dynamic
89	0.83	7.81	0.44
130	1.20	12.40	0.73
225	2.08	21.70	1.10
1090	10.66	105.18	5.38
2146	25.63	296.72	14.16
5010	80.60	605.70	30.82

Fig. 11. Runtime of Herder as the number of parameters increases (left) and the number of LOC increases but keep the number of parameters the same (right), compared to the runtime of a brute-force search and the time to type and cost a single all-dynamic variant. All times are in seconds.

so, which ones) or removing the annotations yields better performance. In 28 out of 30, Herder generates correct recommendations, yielding an accuracy of 93.3%.

Finally, for S4, it is hard to empirically verify Herder's ability to support this scenario without a user study. Still, Figure 10 and our evaluation for S3 show that Herder is effective at both finding performant configurations and directly comparing the performance of two configurations. Consequently, explanations generated by Herder are likely to help users understand the performance bottleneck present in the slower configuration. We leave a user study evaluating Herder's effectiveness for supporting S4 to future work.

6.3 Evaluation of Efficiency

In this subsection we evaluate how Herder scales with the size and complexity of the source program. First, we consider how Herder scales as the number of type parameters in the program increases. Each parameter effectively doubles the size of the search space since it can either be annotated by a static type or not. For our evaluation, we artificially generate a set of programs with an increasing number of parameters. Programs with 2–12 parameters were produced by repeating a small arithmetic function with two parameters 1–6 times; programs with more parameters were produced by copying, pasting, and renaming several functions from the FFT and nbody benchmarks. For each program, we measure the runtime of Herder to type and cost the entire configuration space and produce a recommendation. As baselines for comparison, we also measure the runtime of a corresponding brute-force search of all configurations (that is, generating each configuration and typing/costing each one separately), and also the runtime of typing and costing a single configuration—in this case, the configuration where all parameters are dynamically typed.

The results of the evaluation are shown in the Figure 11 (left). In the table, observe that the brute-force approach scales poorly since the search space grows exponentially with the number of parameters. In contrast, Herder scales approximately linearly when compared to typing and costing a single variant, only taking about 2–4 times as long.

Next we consider how Herder scales as the size of the source program increases. For our evaluation, we generate programs of increasing size but with a fixed number of 4 parameters. We do this by starting with a subset of the scimark FFT benchmark with four parameters, then increase

the length of its core function by repeating its body multiple times. As before, we compare Herder with a brute-force search and also with the time to type and cost the all-dynamic variant.

The results of the evaluation are shown in Figure 11 (right). In the table, observe that all three measurements scale approximately linearly with respect to the size of the program. The runtime of the brute-force approach is approximately 20 times the time to type and cost a single variant. This is as expected since the size of the search space is $2^4 = 16$ and there is some overhead associated with generating the variants and identifying the cheapest configuration. More significantly, observe that Herder also scales linearly and takes 2–3 times the single-variant time. This demonstrates that the size of the program does not induce an unexpected performance hit in our approach.

Together, the evaluations in Sections 6.2 and 6.3 demonstrate that Herder can accurately and efficiently recommend performant migrations of gradually typed Python programs.

7 RELATED WORK

7.1 Static Cost Analysis

There has been substantial work on developing static analyses to infer bounds on the resource usage of programs. For example, automatic cost analyses have been defined over the ML family of languages that can bound the usage of time, memory, or energy [Hoffmann et al. 2017; Hoffmann and Hofmann 2010; Hoffmann and Shao 2015]. Similarly, our work is to statically measure the overhead of a certain resource, namely the number of casts. The methodology used in those works relies on rich type information and uses linear constraint solving to generate the resource bounds. Since gradually typed programs typically lack the rich type information to support these methodologies, our approach is primarily syntax driven. While a main focus of those works is to derive asymptotically tight upper bounds in their analysis for a given program, the main purpose of our cost semantics is to compare the costs of different configurations by observing the worst case behaviors possible for these programs. As Section 6.2 shows, our cost analysis serves this goal well, accurately finding configurations with little overhead from inserted casts.

Our approach of translating source programs into a corresponding cost language was inspired by Danner et al. [2015, 2013]. Their work infers worst-case bounds on the number of evaluation steps to execute a program; we infer worst case bounds on the number of casts performed in the evaluation of a gradual program. We adapt the syntactic generation of approximations from their work in order to produce costs for an appropriate formal model for languages like Reticulated. Unlike theirs, our cost semantics is fully automatic. This means that we can apply our cost analysis to existing Reticulated Python programs without users providing any additional information to help us extract costs.

Relational cost analysis [Çiçek et al. 2017] fulfills a very similar role to variational cost analysis. In relational cost analysis the goal is to reason about the difference in cost between two similar programs or two similar runs of the same program. For example, Çiçek et al. [2016] discuss a relational cost analysis measuring the overhead of a rerun on a program with incremental computation based on changes to the input. While a main goal of relational cost analyses is to relate the cost of two runs of the same program with different inputs, that of variational cost analysis in this work is to relate costs of two similar programs that differ in program structures. However, variational cost analysis still bears much resemblance to the typings in Çiçek et al. [2017], in that sharing in variational typing resembles relational typing and typing choices is similar to using unary typing on unrelated computations. It would be interesting to see how precisely variational techniques can interact with relational cost analysis.

7.2 Gradual Typing Performance

Since Takikawa et al. [2016] reported that sound gradually typed programs can incur massive slowdown when mixing dynamic and static code, there have been several efforts to address the problem.

The Nom programming language [Muehlboeck and Tate 2017] addresses it by designing the static and dynamic semantics for a gradually typed language from the ground up instead of adding gradual typing to an existing language. This helps reduce the overhead of running casts. Nom's design allows programs to gain performance benefits as more types are added to a program.

Bauman et al. [2017] focus on improving performance for Racket by using a tracing JIT compiler, Pycket, with new representations for contracts that eliminate much of their overhead. Since Pycket is a tracing JIT, it can effectively optimize untyped and typed boundaries upon observing them. Though Pycket makes the interaction between typed and untyped code less costly, overhead still remains at certain code boundaries. It would be interesting to see how variational cost analysis and inference can be used to recommend types that eliminate the introduction of typed border crossing in code that iterates heavily, which would in turn help Pycket.

Richards et al. [2017] also help performance by designing intrinsic object contracts for implementations on a virtual machine, allowing the shape checks used by the VM's JIT to act as the runtime type-safety checks for gradual typing. It improves performance by not creating new allocations when a contract is applied to an object that shares the shape of another object that already had the contract applied to it. This design means that the performance of their approach is not determined by the interaction of typed and untyped code, although they can still incur significant memory overhead in certain typing configurations. It would be interesting to see if a variational cost semantics can be created to reason about how different typings cause large memory consumption.

Overall, our approach is orthogonal to these approaches in quite a few ways. First, our methodology is a cost semantics and not a change to a runtime environment. Therefore its goal is not solely optimizing performance but instead reasoning about how types affect performance and can thus be used to support the different migration scenarios outlined in Section 1.2. Second, our semantics is not appropriate for such specialized implementations, where the interaction of typed and untyped code does not incur significant overhead. Instead it is intended for languages that employ the traditional approach of translating gradually typed programs into untyped programs. This makes it appropriate for the implementation of Typed Racket used in practice, which has a traditional contract-based guarded semantics, or for many of the different semantics available for Reticulated. Since many existing gradual languages work by translating typed programs into untyped programs with proxies, we feel that our cost semantics is widely applicable. Finally, the implementation of the cost analysis is relatively simple, with relatively small additions to the type system and cast insertion rules. In contrast, the engineering effort for designing or modifying a JIT compiler may take substantially more work, and codesigning a language's type system and semantics is not an applicable strategy for many existing gradually typed languages.

There are other approaches using type-based techniques to improve program performance. Rastogi et al. [2012] use gradual type inference to help soundly migrate programs toward more precise static types. In their system more precise typing is correlated with performance so they want to infer as many types as possible while preserving soundness. Similarly our system can be adapted to other type inference systems, such as the flow based one they present, so that we can infer many possible most-static types, then find one optimizing static checking and performance.

Finally, confined gradual typing [Allende et al. 2014] introduces constructs to explicitly manage data flow between static and dynamic parts of the program, preventing costly boundary crossings. Our cost analysis can be thought of as a way to reason about what parts of the programs contain these

costly boundary crossings and what type assignments can limit these crossings. This potentially allows our system to help programmers by automatically finding and diagnosing these boundaries.

7.3 Migrational and Variational Typing

The design of the variational cost semantics in Section 5 builds on the technical machinery of migrational typing [Campora et al. 2018], which in turn builds on the prior work on variational typing [Chen et al. 2012, 2014]. This machinery is important to being able to efficiently infer types and perform cost analysis for the entire space of potential migrations. Since migrating gradual types [Campora et al. 2018] focused on the static type safety of gradual programs, it does not contain variational cast insertion, which is necessary in this work in order to measure the costs of the inserted casts. Aside from integrating the cost semantics, the implementation of Herder also required extending that work to support new constructs, such as loops, that are not present in the original calculus but widely used in Python.

8 CONCLUSION

Gradual typing promises the reconciliation of static and dynamic typing. However, a major practical limitation of current implementations is that the interfaces between dynamically and statically typed code can have a huge runtime overhead. Different assignments of type annotations have a significant affect on these costs, but it is hard to predict how to assign types to improve performance. This leaves programmers stuck wondering how to migrate programs to types fulfilling the safety and performance goals they desire for their program.

To address this issue, we have presented a variational cost semantics for gradually typed programs that approximates the runtime costs for all possible type configurations of a program. The cost semantics provides a systematic way to create tools that help programmers identify performant migrations and understand how typing affects performance as they migrate their programs. We have implemented our semantics on top of Reticulated Python in Herder, and our evaluation shows that Herder is effective, efficient, and can be used to aid programmers in many different migration scenarios.

Our approach is amenable to many gradually typed languages in which (partial) type inference is possible, and where inserted casts incur noticeable performance overhead. Overall, this makes variational cost analysis a viable approach to reasoning about the complex interaction of typing and performance during gradual program development. In the future, we will investigate how the ideas in this paper can help address the performance problem in other gradual language implementations, such as Typed Racket.

ACKNOWLEDGMENTS

We thank ICFP and PLDI reviewers, whose reviews have improved both the contents and the presentation of this paper. We also thank the ICFP artifact reviewers who gave many helpful suggestions on how to improve Herder. This work is partially supported by the National Science Foundation under the grant CCF-1750886 and by AFRL Contract FA8750-16-C-0044 (via Raytheon BBN Technologies) under the DARPA BRASS program.

REFERENCES

Esteban Allende, Johan Fabry, Ronald Garcia, and Éric Tanter. 2014. Confined Gradual Typing. *SIGPLAN Not.* 49, 10 (Oct. 2014), 251–270. https://doi.org/10.1145/2714064.2660222

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

John Campora, Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2018. Migrating Gradual Types. In *Proceedings of the 45th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '18).* ACM, New York, NY, USA.

Ezgi Çiçek, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Jan Hoffmann. 2017. Relational Cost Analysis. *SIGPLAN Not.* 52, 1 (Jan. 2017), 316–329. https://doi.org/10.1145/3093333.3009858

Ezgi Çiçek, Zoe Paraskevopoulou, and Deepak Garg. 2016. A Type Theory for Incremental Computational Complexity with Control Flow Changes. *SIGPLAN Not.* 51, 9 (Sept. 2016), 132–145. https://doi.org/10.1145/3022670.2951950

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 (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

Sheng Chen, Martin Erwig, and Eric Walkingshaw. 2016. A Calculus for Variational Programming. In *European Conf. on Object-Oriented Programming (ECOOP) (LIPIcs)*, Vol. 56. 6:1–6:26.

Norman Danner, Daniel R. Licata, and Ramyaa Ramyaa. 2015. Denotational Cost Semantics for Functional Languages with Inductive Types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 140–151. https://doi.org/10.1145/2784731.2784749

Norman Danner, Jennifer Paykin, and James S. Royer. 2013. A Static Cost Analysis for a Higher-order Language. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 25–34. https://doi.org/10.1145/2428116.2428123

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

Martin Erwig and Eric Walkingshaw. 2013. Variation Programming with the Choice Calculus. In *Generative and Transformational Techniques in Software Engineering IV (GTTSE 2011), Revised and Extended Papers (LNCS)*, Vol. 7680. 55–99.

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 (POPL '15)*. ACM, New York, NY, USA, 303–315. https://doi.org/10.1145/2676726.2676992

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards Automatic Resource Bound Analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 359–373. https://doi.org/10.1145/3009837.3009842

Jan Hoffmann and Martin Hofmann. 2010. Amortized Resource Analysis with Polynomial Potential - A Static Inference of Polynomial Bounds for Functional Programs. In *In Proceedings of the 19th European Symposium on Programming (ESOP'10) (Lecture Notes in Computer Science)*, Vol. 6012. Springer, 287–306.

Jan Hoffmann and Zhong Shao. 2015. Automatic Static Cost Analysis for Parallel Programs. In *Proceedings of the 24th European Symposium on Programming on Programming Languages and Systems - Volume 9032*. Springer-Verlag New York, Inc., New York, NY, USA, 132–157. https://doi.org/10.1007/978-3-662-46669-8_6

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

Aseem Rastogi, Avik Chaudhuri, and Basil Hosmer. 2012. The Ins and Outs of Gradual Type Inference. *SIGPLAN Not.* 47, 1 (Jan. 2012), 481–494. https://doi.org/10.1145/2103621.2103714

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

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31.

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages (DLS '08)*. ACM, New York, NY, USA, Article 7, 12 pages. https://doi.org/10.1145/1408681.1408688

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 (POPL '16)*. ACM, New York, NY, USA, 456–468. https://doi.org/10.1145/2837614.2837630

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 964–974. https://doi.org/10.1145/1176617.1176755

Sam Tobin-Hochstadt, Matthias Felleisen, Robert Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. 2017. Migratory Typing: Ten Years Later. In *2nd Summit on Advances in Programming Languages (SNAPL 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.), Vol. 71. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 17:1–17:17. https://doi.org/10.4230/LIPIcs.SNAPL.2017.17

Guido van Rossum, Jukka Lehtosalo, and Łukasz Langa. 2014. PEP 484 – Type Hints. https://www.python.org/dev/peps/pep-0484/#rationale-and-goals

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. (2014), 45–56.

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 (POPL 2017)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/3009837.3009849