Optimizing and Evaluating Transient Gradual Typing

Michael M. Vitousek
Indiana University, USA
mmvitousek@gmail.com

Jeremy G. Siek
Indiana University, USA
jsiek@indiana.edu

Avik Chaudhuri
Facebook Inc., USA
avik@fb.com

Abstract
Gradual typing enables programmers to combine static and dynamic typing in the same language. However, ensuring sound interaction between the static and dynamic parts can incur runtime cost. In this paper, we analyze the performance of the transient design for gradual typing in Reticulated Python, a gradually typed variant of Python. This approach inserts lightweight checks throughout a program rather than installing proxies on higher order values. We show that, when using CPython as a host, performance decreases as programs evolve from dynamic to static types, up to a 6× slowdown compared to equivalent Python programs.

To reduce this overhead, we design a static analysis and optimization that removes redundant runtime checks, employing a type inference algorithm that solves traditional subtyping constraints and also a new kind of check constraint. We evaluate the resulting performance and find that for many programs, the efficiency of partially typed programs is close to their untyped counterparts, removing most of the cost of transient checks. Finally, we measure the efficiency of Reticulated Python programs when running on PyPy, a tracing JIT. We find that combining PyPy with our type inference algorithm results in an average overhead of less than 1%.

CCS Concepts
• Software and its engineering → Object oriented languages; Just-in-time compilers; Runtime environments;

Keywords
Gradual typing, Python, type checks, transient

1 Introduction
Gradual typing enables programmers to gradually evolve their programs from the dynamic typing, which allows flexibility, to static typing, providing security [Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Over the last decade, gradual typing has been of great interest to both the research community [Ahmed et al. 2011; Allende et al. 2013a; Rastogi et al. 2012; Ren et al. 2013; Siek et al. 2015b; Swamy et al. 2014; Takikawa et al. 2012], and to industry, which has introduced several languages with elements of gradual typing, such as TypeScript [Microsoft 2012], Flow [Facebook 2014], and Dart [Google 2011]. Many existing gradually typed languages operate by translating a “surface language” program (i.e. one directly written by a programmer) into a dynamically typed language by erasing types. We refer to the latter as the target language. This type erosion approach is safe in the sense that programs do not elicit undefined behavior as long as the target language is itself safe (for example, because the target language performs runtime checking in every primitive operation). However, we would like gradually typed languages to also be sound with respect to type annotations, that is, a variable annotated with a static type should only be inhabited by values of the appropriate type [Vitousek et al. 2017]. To achieve this kind of soundness, runtime checks are required on the boundaries between statically and dynamically typed code.

1.1 Strategies for Runtime Checks
Several strategies have been used to implement runtime type checking for gradually typed languages, and these strategies are appropriate for different target languages and design goals. The traditional approach in the research literature is to insert casts during translation at the site of every implicit conversion between typed and dynamic code [Allende et al. 2013a; Siek and Taha 2006; Swamy et al. 2014; Tobin-Hochstadt and Felleisen 2006]. At runtime, these casts ensure that values correspond to their expected static type. A cast on a value with first-order type is a constant-time operation that either succeeds or fails, but for higher-order types such as functions and references a cast installs a proxy on the casted value. The proxy ensures that, in the future, the value behaves according to the target type of the cast. This approach is called the guarded strategy [Vitousek et al. 2014].

In recent papers, Vitousek et al. [2014, 2017] identified a number of challenges that make the guarded strategy inappropriate for certain domains. In particular, unless the target language has powerful support for proxies (such as Racket’s
chaperones [Strickland et al. 2012]), interaction between proxied values and foreign functions or target-language code may fail in unexpected ways [Allende et al. 2013b; Van Cutsem and Miller 2013]. Such is the case in Python and these problems were borne out in Reticulated Python, an experimental implementation of gradual typing for Python 3 [Vitousek et al. 2014].

As an alternative, Vitousek et al. [2014] introduced the transient strategy. In this approach, the translation inserts constant-time checks throughout the gradually typed program. These checks do not create proxies but only inspect the type tag [Appel 2007] of the value, failing if it does not correspond to the type constructor the value is statically expected to have (such as int or fun, but not int → int).

To make up for the “transient” nature of these checks, they are inserted throughout program rather than just at the sites of implicit conversions. The translation inserts checks into the program at every function call site, at the entry of every function body (to check that each argument corresponds to its parameter’s type), and at the site of every dereference.

For example, consider the program in Figure 1a, written in a gradually typed language. Here, the makeEq function is a curried equality function on integers with type int → int → bool. It is called to produce the eqFive function on line 12, which is then called at line 14 on the result of calling the iddyn function on a string. Because the result of iddyn has static type ⋆ (representing the dynamic type), this program should pass static typechecking, but a runtime check is needed to detect that at runtime the value being passed into eqFive is actually a string and raise an error.

No matter what strategy is used, in a sound gradually typed language the result should be a runtime error. The transient strategy achieves this goal by inserting checks (such as n :\_ int, which checks that n is an integer) as shown in Figure 1b. Type annotations have been erased, and the bodies of makeEq and its internal function now contain checks at lines 6 and 8 to ensure that their arguments are definitely ints. Similarly, the call to makeEq at line 12 also contains a check, to ensure that the result of the function call is a function. Note that this check does not ensure that the call returns a value of type int → bool—such a check cannot be performed by immediate inspection of the runtime value, but it can verify that the result is a function. It is then up to the function itself to check that it is only passed ints (as it does with the argument checks discussed above), and additional checks are inserted when eqFive is called on lines 13 and 14 to ensure that the result of that call is a bool.

When this program executes, an error will be raised by the check at line 8, because the call to eqFive at line 14 passed in a string. This result is expected and correct; if the error had not arisen, there would be an uncaught error in the body of makeEq’s inner function, as a string would inhabit the int-typed variable n. This error could then (depending on the semantics of equality testing) lead to a confusing, difficult to debug error. As it is, the programmer is simply informed that a type mismatch occurred and where.

Vitousek et al. [2017] showed that this approach supports the open-world soundness property, which states that programs written in a gradually typed language, translated into a dynamic target language, and then embedded in arbitrary code native to that dynamic language, will only “go wrong” due to errors in the native code. The translated, gradually typed program will not be the source of any errors (other than errors caught by transient checks) even in the presence of unmoderated interaction with the “open world.”

1.2 Performance of Transient Gradual Typing

Performance is also of critical concern for gradually typed languages. The runtime checks required for sound gradual typing inevitably impose some degree of runtime overhead, but ideally this overhead would be minimized or made up for by type-based compiler optimizations. Since gradual typing is designed to allow programmers to gradually vary their programs between static and dynamic [Siek et al. 2015a], it is also important that adding or removing individual annotations does not dramatically degrade the program’s performance.

Takikawa et al. [2016] examine the performance of Typed Racket with this criterion in mind by studying programs through the lens of a typing lattice made up of differently typed configurations of the same program. The top of the lattice is a fully typed configuration of the program and the bottom is unannotated, and incrementally adding types moves up the lattice.

Takikawa et al. show that in Typed Racket, certain configurations result in catastrophic slowdown compared to either the top or bottom configurations. This indicates that the guarded semantics incurs a substantial cost when interaction between static and dynamic code is frequent. Many of their benchmarks show mean overheads of over 30× and worst cases of over 100×, which “projects an extremely negative image of sound gradual typing” [Takikawa et al. 2016].

In an effort to establish whether transient gradual typing faces the same problems, Vitousek et al. [2017] performed an initial performance evaluation of Reticulated Python benchmarks and found that overheads (compared to an untyped, standard Python version of the same program) ranged from negligible to over a 5× slowdown. However, this analysis was limited to examining a single configuration, the configuration closest to being fully typed. As shown by Takikawa et al. [2016], this is insufficient to make a strong claim about the overall performance of Reticulated Python.

Greenman and Migeed [2018] evaluated the performance of Reticulated Python using a lattice-based approach similar to that used in evaluating Typed Racket, wherein they found that “the cost of soundness in Reticulated Python is at most one order of magnitude, and increases linearly with the number of type annotations”—a promising result for establishing more acceptable worst-case performance in a sound
def idDyn(a: ⋆ → ⋆): return a

def makeEq(n:int) → int → bool:
  def internal(m:int) → bool:
    return n == m
  return internal

eqFive = makeEq(5)
eqFive(20)
eqFive(idDyn('hello world'))

---

# Program with transient checks

def idDyn(a): return a

def makeEq(n):
  n ⇓ int
  def internal(m):
    m ⇓ int
    return n == m
  return internal

eqFive = makeEq(5) ⇓ fun
eqFive(20) ⇓ bool
eqFive(idDyn('hello world')) ⇓ bool

---

Figure 1. Translation to target language using the transient gradual typing approach.

gradually typed language. A similar result was found for a transient variation of Typed Racket studied by Greenman and Felleisen [2018].

In this paper, we perform a similar analysis of Reticulated Python and come to a similar conclusion, which will serve as a starting point for the remainder of our work. We found that the cost of transient gradual typing increases as the number of type annotations grows, resulting in a predictable linear degradation of performance to a worst case $6 \times$ overhead.

1.3 Reducing the Burden of Pervasive Checks

The transient approach inserts checks throughout the program, but not all checks are necessary for the program to be sound because some checks may be redundant and always succeed. To remove unnecessary checks, we perform type inference on the program after checks have been inserted. Our inference algorithm is based on those of Aiken and Fähndrich [1995] and Rastogi et al. [2012] and uses subtyping constraints as well as new check constraints, generated by transient checks. We prove that our algorithm can soundly remove unnecessary checks in a transient calculus similar to that of Vitousek et al. [2017].

We modified Reticulated Python to support this optimization and measured its performance, again sampling from the typing lattices at all levels. With redundant checks removed, the linear increase in execution times disappears, resulting in the fully typed configurations displaying negligible overhead and a 6% average overhead over all sampled configurations.

1.4 Transient Gradual Typing on a Tracing JIT

While this analysis removes many checks statically, the nature of transient checks suggests that they could also be dynamically optimized away by a JIT. Fortunately, there is a tracing JIT for Python 3, PyPy [Bolz et al. 2009]. Reticulated Python compiles to standard Python 3, so it is suitable to use with PyPy. We found that gradually typed programs running on PyPy displayed much less overhead than the same configurations running on CPython—the average overhead over all configurations was 3% with PyPy compared to 2.21× with CPython, suggesting that PyPy is able to optimize away most of the overhead of transient checks. Some benchmarks still incurred a linear increase in time as types were added, but to a lesser degree than with CPython (with a worst case overhead of 2.61×). Our optimization proved to be complimentary to using PyPy as a host, and combining them resulted in a runtime that showed negligible detectable overhead compared to running untyped programs on PyPy.

1.5 Contributions

In this work, we measure the performance of transient gradual typing in Reticulated Python and design techniques to improve it. Our contributions are:

- We analyze the performance of Reticulated Python programs across their typing lattices, finding an average overhead of 2.21× and worst case of 6× (Section 2).
- We develop a type inference optimization for reducing the number of checks needed by the transient approach (Section 3).
- We implement this optimization in Reticulated Python and show that it reduces the average overhead to just 6% across all typing lattices (Section 4).
- We analyze both the unoptimized and optimized versions of Reticulated when running under a tracing JIT, and find that it performs very well, especially in combination with our optimization (Section 5).

Section 6 discusses related work, and Section 7 concludes.
2 Performance of Transient Gradual Typing

Before investigating approaches to improve the performance of transient gradual typing, we first establish the performance characteristics of Reticulated Python across the typing lattice [Takikawa et al. 2016]. Vitousek et al. [2017] implemented a blame-tracking technique for transient gradual typing in Reticulated Python, allowing programmers to trace runtime type errors back to the crossing-points between static and dynamic that led to the error, but for this analysis we disable blame tracking. Conducting the evaluation with blame tracking is important future work.

2.1 Experimental Methodology

We selected ten Python 3 programs and translated them to Reticulated Python by inserting type annotations. These benchmarks are mostly drawn from the official Python benchmark suite,1 with several drawn from the analysis of Takikawa et al. [2016] and translated from Racket to Python. In most cases, the resulting Reticulated programs are fully annotated with static types. However, even with a fully annotated program, the Reticulated type checker can assign expressions the type Any (the dynamic type), such as an if-then-else expression where the branches have different types; we did not attempt to guarantee that such cases do not arise.

To examine the typing lattice for a benchmark, we first count the number of type constructors that appear in the program’s annotations. We call this the type weight of a program. For example, the presence of the type List[int] in a program’s annotations adds 2 to its weight, and Callable[[int], bool] (which is the Reticulated representation of the type int → bool) adds 3. We then divide the type weight into a maximum of 100 intervals: a program with a type weight of 300 would have intervals [0, 3), [3, 6), . . . , [297, 300). Programs with a total type weight of less than 100 naturally have fewer than 100 intervals. For each interval, we randomly erase type annotations and replace them with Any, until the program’s type weight falls within the interval. This process can “dynamize” types underneath type constructors; both Any and List[Any] are possible types that could be generated from an original type annotation List[int]. Each partially dynamized program is a configuration sampled from the typing lattice at the level corresponding to its type weight. We generate ten configurations per interval, plus a single fully typed configuration consisting of the original program, for a maximum of 1001 configurations.

Each configuration was executed on an Intel Core i3-4130 CPU with 8GB of RAM running Ubuntu Server 14.04. Configurations were executed on CPython 3.4.3 and runtimes were recorded. In order to account for variance in the execution time of the benchmarks, each configuration was executed multiple times and the results averaged. The number of test executions undergone by each sample varied by the runtime of each execution, to keep the overall result generation time tractable; the lowest was go, with 5 executions, while the highest was sieve, with 50,000. The snake benchmark was executed 10 times, the sieve benchmark 3,000, and the rest executed 35 times. Each configuration was tested within its own process to ensure that one configuration could not affect the execution of another, while each execution of an individual configuration was executed by the same process.

This sampling methodology differs from that used by Greenman and Migeed [2018] in that their evaluation is exhaustive for programs with up to $2^{21}$ total possible configurations, and after that the number of sampled configurations scales in proportion to the number of function and class definitions. Further, rather than generating configurations at a per-type-constructor granularity, they generate configurations at the level of individual functions, methods, or sets of class fields. In this evaluation, the number of evaluated configurations scales by type weight, a measure that corresponds to the per-constructor level of granularity, but has a much lower maximum number of configurations as a result of using consumer hardware rather than a computing cluster as a testbed. However, both overall evaluations arrive at similar conclusions, as discussed below.

2.2 Results

Figure 2 shows the execution times for configurations across the typing lattice for each benchmark. Each graph corresponds to one benchmark, and each red circle in the graph represents the average execution time of one configuration. The dashed line marks the execution time of the untyped version of the benchmark in standard Python 3. (The blue triangles show the performance of optimized configurations, discussed below in Section 4). Moving from left to right moves up the lattice from untyped to typed, execution time is shown on the left y axis, and relative overhead compared to the untyped program is shown on the right y axis—higher y coordinates indicates slower performance, and an overhead of exactly 1× means that the configuration took exactly as long to execute as the untyped version of the program.

Performance degrades as types are added because changing an annotation from Any to a static type results in checks being inserted. The graphs of pystone, snake, and others display a linear degradation because each check is executed approximately the same number of times when the configuration runs. Graphs with greater variance, such as that of spectral norm, arise when some checks are executed more often than others, so different configurations at the same point in the lattice perform differently depending on which annotations are dynamized. The configurations from meteor_contest form two large clusters: the parameter fps of the solve function has type List[List[List[Set[int]]]]; solve loops over the second and third dimensions of this value and every iteration of a loop includes a check if the

---

1https://github.com/python/performance
loop’s target is typed, and so replacing fps’s annotation with Any or List[Any] dramatically reduces the amount of time spent performing transient checks.

Over the full typing lattices of all benchmarks, Reticulated Python incurs an average overhead of 2.21× compared to the untyped versions of the benchmarks. Typically the slowest configurations are the ones with the highest type weight: fully typed configurations have an average overhead of 3.63×. The slowest configuration is from nbody at 5.95×.

Overall, these results further confirm the conclusions of Greenman and Migeed [2018] and Greenman and Felleisen [2018] that the performance cost of transient gradual typing is significant, but generally predictable across the lattice and remaining within an order of magnitude. However, the overheads seen here are still prohibitive for many applications. The remainder of this paper will develop two techniques for improving this situation: performing an ahead-of-time optimization, and using a more suitable runtime system.
3 Optimizing Transient Gradual Typing

To improve Reticulated Python's performance, we aim to reduce the number of checks while preserving those required for soundness. The basic idea of transient gradual typing is to use pervasive runtime checks to verify that values correspond to their expected static types. With the transient approach, type annotations are untrusted: they do not provide information to be relied on, but rather are claims that must be verified. Therefore, to reduce the runtime burden of transient gradual typing, we move this verification from runtime to compile time wherever possible. We do so by using type inference to determine when types can be trusted and do not need runtime verification.

Our inference process is based on the approaches of Aiken and Wimmers [1993] and Rastogi et al. [2012], using subtyping constraints and also a new form of constraint, the check constraint. To determine which checks are redundant, our inference algorithm occurs after transient checks have already been inserted, because the existence of a check in one part of a program can allow checks elsewhere to be removed. Check constraints let the system reason conditionally about checks, and they express the idea of transient checks: the type of the check expression being checked \( e \) is constrained to be equal if, when solved for, the type of \( e \) corresponds to the type tag \( S \) (for example, if the type of \( e \) is solved to be \( \alpha \rightarrow \beta \) and \( S \) is fun). If that is not the case, for example if \( S \) is fun and the type of \( e \) is solved to be \( \star \), then the type of the overall check \( e \| S \) is constrained to be the most general type that corresponds with \( S \) (in this example, \( \star \rightarrow \star \)).

3.1 Overall Approach

We generate sets of check constraints and subtype constraints from programs and find a solution that maps type variables to types, and then remove redundant checks. Our approach is as follows:

- Perform transient check insertion as described by Vitousek et al. [2017].
- Assign a unique type variable to every function argument, return type, and reference in the program.
- Perform a syntax-directed constraint generation pass.
- Solve the constraint system to obtain a mapping from type variables to types.
- Using this mapping, perform a syntax-directed translation to the final target language. For each check in the program, if the inferred type of the term being checked and the tag it is checked against agree, remove the check, otherwise retain it.

As an example, we return to the program shown in Figure 1a, which shows a curried equality function written in a gradually typed language and which should pass static type-checking but fail due to a transient check at runtime. Figure 3a shows the result of this program after the first phase of our optimizing translation. In this phase, transient checks have been inserted exactly as in Figure 1b, but instead of the programmer’s type annotations being erased, they have been replaced by type variables \( \alpha, \beta, \gamma, \delta, \epsilon, \zeta \).

Our system infers types (which may be entirely different from the programmer’s annotations) for these type variables by generating subtyping constraints and special check constraints. Check constraints are generated by transient checks, and serve to connect the type of the checked expression with the type it is used at after the check. For example, at line 12, the result of \( \text{makeEq}(5) \) has type \( \delta \), and is then checked to ensure that it is a function \( (\| \text{fun}) \). The type of the result of this check, and therefore the type of \( eq\_\text{fun} \), is \( \eta \rightarrow \theta \), where \( \eta, \theta \) are fresh type variables. This type is linked to \( \delta \) by a check constraint \( (\delta: \text{fun}) = \eta \rightarrow \theta \), which can be read as “if \( \delta \) is solved to be a function, then its solution is equal to \( \eta \rightarrow \theta \).”

We use check constraints rather than equality constraints [Hindley 1969; Milner 1978] because the same variable can be checked against many different types at different points in the program. Check constraints are only generated by transient checks where the checked type tag corresponds to a non-base type, because constraints of the form \( (\gamma: \text{int}) = \text{int} \) (as would be generated on line 6) add no new information to the system: the type on the right will be int whether \( \gamma \) is solved to be int or not.

Subtyping constraints are also generated from the program. For example, because the call to \( \text{makeEq} \) on line 12 has an integer argument, it generates the constraint \( \text{int} <: \gamma \), meaning that \( \gamma \) is constrained to be a supertype of int. The full set of constraints for this example is:

\[
\{ \alpha <: \beta, \epsilon \rightarrow \zeta <: \delta, \text{bool} <: \zeta, \text{int} <: \gamma, (\delta: \text{fun}) = \eta \rightarrow \theta, \text{int} <: \eta, \beta <: \eta \}
\]

We then solve this constraint set to obtain a mapping from each variable to a single non-variable type. The only subtyping constraint on \( \delta \) is that \( \epsilon \rightarrow \zeta <: \delta \), so we determine that \( \delta \) must be a function and that \( \epsilon \rightarrow \zeta = \eta \rightarrow \theta \). This, combined with the fact that both int and (transitively) str must be subtypes of \( \eta \) due to the calls on lines 13 and 14, means that the only solution for \( \eta \) and \( \epsilon \) is \( \star \) (the dynamic type). In all, the solution we find for this constraint set is \( \alpha = \beta = \text{str}, \gamma = \text{int}, \delta = \star \rightarrow \text{bool}, \epsilon = \eta = \star, \) and \( \zeta = \theta = \text{bool} \).

Some of the transient checks in Figure 3a verify information that the constraint solution has already confirmed. For example, the check at line 6 verifies that \( n \) is an integer— but \( n \)’s type \( \gamma \) was statically inferred to be integer, and so this check is not needed. However, the check at line 8, which verifies that \( m \) is an integer, is not redundant: the type \( \star \) was inferred for \( m \)’s variable \( \epsilon \). In fact, this check is needed for soundness because it will fail with a string on line 14. The final program, with redundant checks removed and all annotations erased, is listed in Figure 3b.

For the purposes of this example we do not include constraints based on potential interaction with the open world.
Figure 3. Stages of optimized transient compilation for the program shown in Figure 1.

variables $x, y$, numbers $n \in \mathbb{Z}$
type vars. $\alpha, \beta, \gamma$

$\lambda^T_d$ exprs. $d ::= d \cdot S \mid x \mid n \mid d + d \mid \lambda(x: \alpha) \rightarrow \beta. d \mid d \cdot d \mid \text{let } x = d \text{ in } d \mid \text{ref}_d \alpha. d \mid !d \mid d := d$

type tags $S ::= \text{int} \mid \text{fun} \mid \text{ref}$

Figure 4. The $\lambda^T_d$ transient check calculus (based on Vitousek et al. [2017]).

If this program were to be visible to the open world and potentially used by untyped Python clients, we would need to generate the additional constraints $\star <: \alpha$ and $\star <: \gamma$ in order to maintain open-world soundness, because the open world could pass arbitrary values into these functions [Vitousek et al. 2017].

3.2 Constraint Generation with Check Constraints

In this section we describe our approach to generating type constraints for programs in a transient calculus $\lambda^T_d$. Programs in $\lambda^T_d$ are not the surface programs written by the programmer, and $\lambda^T_d$ is not a gradually typed language. Instead, $\lambda^T_d$ programs are the result of translating programs from a gradually typed surface language. This translation inserts transient checks, written $d \cdot S$, at expressions at their elimination forms, uses of the type consistency relation (which replaces type equality in gradual type systems), and on parameters at function entry points Vitousek et al. [2017]. This approach follows that of the cast calculi in guarded gradual typing [Herman et al. 2007; Siek and Taha 2006; Wadler and Findler 2009], which similarly are the result of a translation from a gradually typed surface language into a target language where runtime enforcement of types is made explicit.

Figure 4 shows the syntax for $\lambda^T_d$, which includes transient checks $d \cdot S$, which at runtime compare the runtime type information of the evaluation of $d$ with the type constructor (or type tag) $S$. The $\lambda^T_d$ calculus also gives type variables to function parameter positions, function returns, and mutable references: we assume that these variables are unique and were inserted during translation to $\lambda^T_d$, and the primary work of optimizing $\lambda^T_d$ programs consists of inferring types $T$ (see Figure 6) for these variables.

To infer types for these type variables, we first generate constraints using the syntax-directed rules defined in Figure 5, in the style of Aiken and Fähndrich [1995]. These rules generate sets $\Omega$ of constraints $C$ over types $A$, also defined in Figure 5. Types $A$ are not inductively defined—function and reference types can only contain type variables or $\star$, not arbitrary types. Subtyping constraints are generated from function and reference introduction and elimination sites, to ensure that any solution found for these variables is well-typed.

For example, consider the following program:

$$(\lambda(x: \alpha) \rightarrow \beta. x) 42$$

The application rule would generate the constraint $\text{int} <: \alpha$, and the function rule would generate $\alpha <: \beta$, so any solution to $\alpha$ and $\beta$ must allow the integer 42 to be passed into $\alpha$ and the type of $\alpha$ to be returned through $\beta$.

The rule for transient checks differs from the others. First, there is the question of what constraint type to give the result of a check. Consider the program $\lambda(x: \alpha) \rightarrow \beta. (x \cdot \text{fun})$. In the body of the function $x$ has type $\alpha$, but the type of the check expression $x \cdot \text{fun}$ ought to be a function, because the check will fail at runtime if $x$ is not a function. The check cannot, however, specify argument and return types. Therefore the type of $x \cdot \text{fun}$ is a function whose argument and return types are fresh type variables. This type is obtained using the $\triangleright_S$
leaf types \[ V ::= \alpha \mid \star \]
constraint types \[ A ::= V \mid \text{int} \mid \text{ref} \mid V \rightarrow V \]
constraints \[ C ::= A <:: A \mid (A:S) = A \]
constraint sets \[ \Omega \in \mathcal{P}(C) \]

\[ A \triangleright \Delta A \]

\[ V_1 \rightarrow V_2 \triangleright_{\text{fun}} V_1 \rightarrow V_2 \]
\[ \alpha \triangleright_{\text{ref}} \beta \rightarrow \gamma \quad \text{with } \beta, \gamma \text{ fresh} \]
\[ \text{ref} \ V \triangleright_{\text{ref}} \text{ref} \ V \]
\[ \text{int} \triangleright_{\text{int}} \text{int} \]
\[ V \triangleright_{\text{int}} \text{int} \]

\[ \Gamma \vdash d : A ; \Omega \]
\[ \Gamma \vdash d : A_1 ; \Omega \quad A_1 \triangleright \Delta A_2 \]
\[ \Gamma \vdash d \parallel \Delta S : A_2 ; \Omega \cup \{ (A_1:S) = A_2 \} \]
\[ \Gamma, x : \alpha \rightarrow d : A ; \Omega \]
\[ \Gamma \vdash \lambda(x: \alpha) \rightarrow \beta. \ d : \alpha \rightarrow \beta ; \Omega \cup \{ A <: \beta \} \]
\[ \Gamma \vdash d_1 : V_1 \rightarrow V_2 ; \Omega_1 \quad \Gamma \vdash d_2 : A ; \Omega_2 \]
\[ \Gamma \vdash d_1 \rightarrow d_2 : V_2 ; \Omega_1 \cup \Omega_2 \cup \{ A <: V_1 \} \]
\[ \Gamma \vdash d : A ; \Omega \]
\[ \Gamma \vdash \text{ref}_\alpha d : \text{ref} \alpha ; \Omega \cup \{ A <: \alpha \} \]
\[ \Gamma \vdash d_1 : \text{ref} \ V ; \Omega_1 \quad \Gamma \vdash d_2 : A ; \Omega_2 \]
\[ \Gamma \vdash d_1 := d_2 : \text{int} ; \Omega_1 \cup \Omega_2 \cup \{ A <: V \} \]
\[ \Gamma \vdash d_1 : \text{int} ; \Omega_1 \quad \Gamma \vdash d_2 : \text{int} ; \Omega_2 \]
\[ \Gamma \vdash d_1 + d_2 : \text{int} ; \Omega_1 \cup \Omega_2 \]
\[ \Gamma(x) = A \]
\[ \Gamma \vdash d : \text{ref} \ V ; \Omega \]

\[ \Gamma \vdash x : A ; \emptyset \quad \Gamma \vdash n : \text{int} ; \emptyset \quad \Gamma \vdash \!d : V ; \Omega \]

\textbf{Figure 5.} Syntax and constraint generation for \( \lambda^\parallel_d \)

\textbf{relation in Figure 5, where } S \text{ is the type tag checked against (in this case fun). If the type on the left has } S \text{ as its constructor, the type on the right is the same, but if it is a variable the right-hand side is a new type that corresponds to } S \text{ but is otherwise inhabited by fresh variables.}

Checks do not generate subtype constraints: checking that something with type } \alpha \text{ is a function should not introduce the constraint } \alpha <:: \beta \rightarrow \gamma, \text{ because the solution to } \alpha \text{ might not actually be a function. For example, if all values that flow into a variable with type } \alpha \text{ are integers, then at runtime this check will fail, which is an acceptable behavior for gradually typed programs. However, if only functions inhabit } x, \text{ then the argument and return types of those functions must be equal to } \beta \text{ and } \gamma \text{ respectively. Check expressions instead generate check constraints, a form of conditional constraint written } (A_1:S) = A_2, \text{ which constrain the solution for } A_2 \text{ so that, if the solution of } A_1 \text{ corresponds to the type tag } S, \text{ then } A_1 = A_2.\]

The constraint generation system in Figure 5 is sufficient to find a solution if the program does not interact with external code (a closed world), but not if the program can interact with code that may not know about or respect the types it expects. Vitousek et al. [2017] showed that transient check insertion ensures safety in open world contexts, but optimizing transient programs based only on internal information could result in the deletion of checks critical to preserving open-world safety. Fortunately, open-world soundness can be achieved, following Rastogi et al. [2012], by adding additional constraints to the system that bound the top-level types of the program with \( \star \) as an upper bound. This allows our analysis to be modular: individual modules can be optimized in isolation by making pessimistic assumptions about what kinds of values flow from one module to another.

\textbf{3.3 Computing Constraint Solutions}

We compute constraint sets } \Omega \text{ to solutions that map type variables } \alpha \text{ to types } T, \text{ as defined in Figure 6. We use a standard approach to subtype-constraint-based inference based on the algorithms of Aiken and collaborators [Aiken and Fähndrich 1995; Aiken and Wimmers 1993; Aiken et al. 1994], but with the addition of check constraints, requiring constraint sets to be solved incrementally in a similar manner to the conditional constraints of Pottier [2000]. This inference algorithm is also similar to that of Chaudhuri et al. [2017], and is described in full detail in our technical report.\textsuperscript{2}

\textbf{3.4 Check Removal}

When a constraint set } \Omega \text{ is solved the types in } \sigma \text{ can be relied on because they were inferred directly from the program, rather than declared as a specification by the programmer. They may be more or less precise, or entirely different, than the programmer’s annotated types. When these types align (i.e. when the inferred type is more or equally precise than the annotated type), any runtime type check that enforces this type must be redundant, and can be removed. On the other hand, when the inferred type is less precise, the type

\textsuperscript{2}https://arxiv.org/abs/1902.07808
check is meaningful and must be preserved, and when the types are unrelated, this corresponds to a runtime type check that will always fail. Using these properties, $\sigma$ allows for $\lambda$ programs to be translated to a final target language that is untyped but retains runtime type checks where necessary. Since this step is an optimization, in order to retain semantic equivalence to the original program any checks that will always fail are preserved, but in practice a warning can be raised to the programmer, and Reticulated Python’s implementation of this algorithm does so. This translation is straightforward and is given in our technical report, as a proof of correctness for this optimization, which guarantees that any check that would be removed by this process could never evaluate to a runtime failure if it was retained.

4 Performance of Optimized Transient Gradual Typing

In this section we apply the above approach to optimize Reticulated Python, and summarize Reticulated’s performance characteristics when running on CPython. This required expanding the type system and constraint generation of Section 3 to handle Python features such as objects and classes, data structures such as lists and dictionaries, bound and unbound methods, and variadic functions. In addition, the constraint generation includes polymorphic functions and intersection types, although they only occur in the pre-loaded type definitions for Python libraries and built-in functions.\(^3\)

Figure 2, as previously discussed, shows the execution time and overheads for optimized configurations from the typing lattice of each benchmark when executed using CPython. Optimized configurations are shown as blue triangles. Performance is dramatically improved compared to the unoptimized (red circle) configurations. In several benchmarks, the overhead is entirely eliminated because the optimization is able to delete nearly every check in every configuration from the typing lattice. For these results, we make the closed world assumption for the benchmarks.

The suffixtree benchmark, which Takikawa et al. [2016] tested in Typed Racket and found overheads of up to 105×, also performs worse than other benchmarks in Reticulated Python after optimization: although it has negligible overhead in configurations with high type weight, some configurations with intermediate type weight still have an overhead of over 2×. This is because suffixtree, unlike the other benchmarks, cannot be fully statically typed using Reticulated Python’s type system: the version with the highest type weight, used to generate the other samples, still includes a function, node_follow, whose return type cannot be given a static type by the Reticulated Python type system or the inference process, because it can return either functions or booleans. Because node_follow’s return values flow into statically typed code, checks will be needed. In most configurations, checks occur when the result of node_follow passes into a statically typed function, and checks can be removed from the rest of the program. However, some configurations allow the dynamic values to flow further into the program before encountering a check; the dynamicity has “infected” more of the program and degraded performance.

Figure 7 summarizes the performance results for both optimized and unoptimized Reticulated Python under CPython. For each benchmark, the table shows the mean overhead ("Mean"), maximum overhead ("Max"), and the overhead for the fully typed (or nearest to fully typed) configuration ("Static"). These figures are given both for the original unoptimized approach ("Unopt") and with our optimization ("Opt"). Without optimization, the average of all configurations from each benchmark meets the cutoff of 3-deliverability as suggested by Takikawa et al. [2016], meaning that their overheads were 3× or less, but most static cases in the benchmarks are not 3-deliverable. Our optimization dramatically improves the results: not only are all configurations 3-deliverable, but all fully typed configurations pass the stricter cutoff of 1.25-deliverability, with slowdowns of 25% or less over the original untyped program. In all, we found that with the optimization, the average overhead across the typing lattices of all benchmarks was only 6%.

5 Performance on PyPy, a Tracing JIT

The analysis described in Section 3 and evaluated in Section 4 removed checks when they can be statically guaranteed to never fail, which suggests that a dynamic analysis could accomplish the same task. We examined this question using a tracing JIT-based VM for Python 3, PyPy [Bolz et al. 2009].

Our benchmarking methodology for PyPy is nearly identical to our approach for CPython, described in Section 2.1. We tested the same configurations under PyPy that were tested...
with CPython; we did not re-sample the lattice. As above, each configuration was tested with a separate instance of the PyPy executable to ensure that one configuration could not warm up the JIT to more efficiently execute another. Also as above, each configuration was run multiple times to account for variance and to attempt to ameliorate the effect of JIT warmup. In one case, we increased the iteration count of pystone (the quickest benchmark in our suite) by 100× to ensure the JIT was warmed up.

Figure 8 shows the typing lattices for our benchmarks when run on PyPy, both with the standard and optimized transient approaches. Figure 9 summarizes these results. Without static check removal, PyPy’s performance varies but is almost always better than CPython’s relative to the baseline performance of the untyped benchmarks. PyPy performs on average 4.3× better than CPython on its own when comparing untyped benchmarks, so when examining the performance cost of transient gradual typing, we always compare
6 Related Work

Gradual type systems. In our work, checks are removed when we infer that the types they verify can be trusted. This is suggestive of the strict confined gradual typing of Allende et al. [2014], which allows programmers to restrict types such that their inhabitants must never have passed through dynamic code (indicated by ↓T) or will never pass through dynamic code in the future (↑T). The type system verifies that terms with type ↓T originated from an introduction form for values of that type. We believe that this information could be used to remove transient checks.

Our approach is also related to concrete types [Richards et al. 2015; Wrigstad et al. 2010], which are inhabited only by non-proxied values (using the guarded cast strategy). Concrete types have limited interoperability with dynamic types and with like types, which are the types of values which may or may not be proxied. While the formulation of concrete vs. like types given by Wrigstad et al. [2010] is appropriate for the guarded semantics, splitting types into those which can be statically relied on and those which need runtime verification is similar to our approach.

Performance analysis for gradual typing. Takikawa et al. [2015, 2016] introduced the typing lattice to evaluate gradually typed languages and found that the overhead of Typed Racket in some configurations was high enough to threaten the viability of gradual typing. Greenman and Migeed [2018] analyzed Reticulated Python programs across the typing lattice using a sampling-based methodology similar to ours, and reported performance results similar to those that we report in Section 2; this work was confirmed by Greenman and Felleisen [2018] in a transient variant of Typed Racket. Muehlboeck and Tate [2017] designed a gradually typed language with nominal object types and without structural types or functions and examined its performance across the lattices of several benchmarks, finding negligible overhead.

Kuhlenschmidt et al. [2019] build an ahead-of-time compiler for a gradually typed language Grift, which uses both
the traditional guarded strategy for gradual typing and a monotonic semantics in which casts are permanently attached to values [Siek et al. 2015c]. Griff’s compiler is statically highly tuned for efficiently executing runtime checking and proxying, and shows high performance in the fully statically typed case and overheads of up to around 3× (compared to the dynamic case) when partially typed.

Richards et al. [2017], discussed above in Section 5, use a monotonic contract system in combination with a specialized VM to achieve an average slowdown of 6.6%, with average worst-case configurations showing an average slowdown of 10.2%. By comparison, an early version of Reticulated Python [Vitousek et al. 2014] implemented a similar monotonic system (that of [Siek et al. 2015c]) and found overheads of greater than 10×, suggesting that modifying VMs to be aware of gradual runtime enforcement can pay dividends.

Roberts et al. [2019], likewise discussed above, use the Graal VM as a host for their implementation of Moth, which uses transient gradual typing and a carefully tuned representation of transient checks that allows the JIT compiler to effectively optimize away many transient checks, resulting in dramatically improved performance over an unoptimized baseline, and an overall average overhead of 5% over the untyped versions of their programs.

**Type inference.** Aiken and Wimmers [1993] generalize equational constraints used in standard Hindley-Milner type inference [Hindley 1969; Milner 1978] to subtyping constraints. Their type system includes union and intersection types and generates systems of constraints that may be simplified by rewriting. Aiken and Fähndrich [1995] specialize this approach to determine where coercions are needed in dynamically typed programs with tagging and untagging in order to optimize Scheme programs. This approach is similar to ours, except that while our type system is much simpler and does not include untagged types and values, transient checks must generate check constraints. Check constraints are similar to the conditional constraints of Pottier [2000], but rather than allowing arbitrary implications, check constraints reason exclusively about type tags.

Soft type systems [Aiken et al. 1994; Cartwright and Fagan 1991] use a similar approach to integrating static and dynamic typing with type inference, but with a different goal: to allow programs written in dynamically typed languages to leverage the benefits of static typing. Soft type systems use type inference to determine where runtime type checks must be inserted into dynamically typed programs so they are well-typed in some static type system. Cartwright and Fagan [1991] reconstruct types for their language and determine where checks are needed using circular unification on equality constraints [Weis 1987] over an encoding of the supertypes of each type in the program which factors out subtyping [Rémy 1989]. Aiken et al. [1994] adapt subset constraint generation to soft typing with similar results. The runtime checks, or narrowers, used in soft typing are similar to transient checks: a narrower checks that a value corresponds to a specific type constructor. The value is returned unmodified if so, and an error is raised if not. Narrowers serve a different purpose than checks, however: narrowers let programs written in dynamically typed languages pass static typechecking for optimization, while checks enforce the programmer’s claims about the types of terms.

The constraint system used by Flow’s type inference system, described for the Flowcore calculus by Chaudhuri et al. [2017], reasons about tags and tagchecks in order to find sound typings for JavaScript programs, though it relies on and trusts type annotations at module boundaries. Likewise Guha et al. [2011] relate type tags and types similarly to our approach and their tagchecks are equivalent to transient checks, but their goal is to insert the tagchecks needed to type function bodies with respect to their annotations, rather than to detect violations of those annotations by callers.

Rastogi et al. [2012] optimize gradually typed programs by inferring more precise types for program annotations, while preserving the program’s semantics. Our approach to ensuring interoperability is based on theirs: visible variables in the overall type of a program and their co- or contravariant positions encodes escape analysis, and solutions that can soundly interoperate with arbitrary code can be generated by adding constraints on these variables. Our constraints, however, are based on subtyping rather than on consistency, because our constraints arise from checks and from elimination forms rather than from casts, which are appropriate for guarded gradual typing rather than transient.

**7 Conclusions**

Gradual typing allows programmers to combine static and dynamic typing in the same language, but allowing interaction between static and dynamic code while ensuring soundness incurs a runtime cost. We find that Reticulated Python, a gradually typed programming language using the transient strategy for runtime checks, shows an overhead of up to 6× compared to untyped Python programs. To reduce this overhead, we use a static type inference algorithm based on subtyping and check constraints to optimize programs after transient checks are inserted. This allows many redundant checks to be removed. We evaluate the performance of this approach with an implementation in Reticulated Python and find that performance across the typing lattices of our benchmarks improves to nearly the efficiency of the original programs—a very promising result for the practicality of gradual typing. Finally, we re-analyze our Reticulated Python benchmarks using PyPy, a tracing JIT, as a backend, and find that it produces good performance even without type inference, and that it displays overhead of less than 1% on average, and 23% in the worst case we studied, when used in combination with our static optimization.
Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. 2017. Big Types in Little Runtime: open world soundness and collaborative blame for gradual type systems. In POPL.

Philip Wadler and Robert Bruce Findler. 2009. Well-typed programs can’t be blamed. In ESOP.
