Sound Gradual Typing is Nominally Alive and Well

FABIAN MUEHLBOECK, Cornell University, USA
ROSS TATE, Cornell University, USA

Recent research has identified significant performance hurdles that sound gradual typing needs to overcome. These performance hurdles stem from the fact that the run-time checks gradual type systems insert into code can cause significant overhead. We propose that designing a type system for a gradually typed language hand in hand with its implementation from scratch is a possible way around these and several other hurdles on the way to efficient sound gradual typing. Such a design process also highlights the type-system restrictions required for efficient composition with gradual typing. We formalize the core of a nominal object-oriented language that fulfills a variety of desirable properties for gradually typed languages, and present evidence that an implementation of this language suffers minimal overhead even in adversarial benchmarks identified in earlier work.

CCS Concepts: • **General and reference** → **Performance**; • **Software and its engineering** → **Formal language definitions**; **Runtime environments**; Object oriented languages;

Additional Key Words and Phrases: Gradual Typing, Nominal, Immediate Accountability, Transparency

ACM Reference Format:
Fabian Muehlboeck and Ross Tate. 2017. Sound Gradual Typing is Nominally Alive and Well. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 56 (October 2017), 30 pages. https://doi.org/10.1145/3133880

1 INTRODUCTION

Sound gradual typing is the idea that typed and untyped code can be mixed together in a single language in such a way that the typed code is able to execute as if there were no untyped code. This means that typed code can rely on type soundness to enable type-driven optimizations.

The basic intuition of how to achieve sound gradual typing is relatively simple: we must protect the guarantees obtained through static type-checking by inserting run-time checks at locations in the code where values flow from untyped components to typed components. If a value from untyped code fails to have its expected type at run time, an exception is thrown. Thus, the statically checked components of the program can assume all values passed to them are well-typed.

Painted this way, the picture leads us to expect that gradual typing may incur some overhead for those inserted checks, proportional to the number of times we transition from untyped to typed parts of the program. In the optimal scenario, these checks are infrequent and efficient, and thus the overall cost of gradual typing is low and can be easily estimated and planned for by analyzing the level of interaction between typed and untyped parts of the program. Furthermore, typed code can be optimized in ways untyped code cannot, so one would expect performance to smoothly improve as types are added to a code base. However, we are not aware of any existing proposal for a sound gradually typed language which has such performance behavior and reasonable performance for

Authors' addresses: Fabian Muehlboeck, Cornell University, Ithaca, NY, 14850, USA, fabianm@cs.cornell.edu; Ross Tate, Cornell University, Ithaca, NY, 14850, USA, ross@cs.cornell.edu.

2475-1421/2017/10-ART56
https://doi.org/10.1145/3133880

fully untyped code. In fact, Takikawa et al. [2016] measured extreme and unpredictable dips in performance for programs consisting of both typed and untyped code. These measurements were made on their current proposal for sound gradual typing for Racket [Tobin-Hochstadt and Felleisen 2006], but they argue that no other system provides convincing reasons for why it should perform significantly better.

In this paper, we present a way to achieve efficient sound gradual typing. The core component of this approach is a nominal type system with run-time type information, which lets us verify assumptions about many large data structures with a single, quick check. The downside of this approach is that it limits expressivity, particularly with respect to structural data. Nonetheless, we argue that one should be able to build useful programming languages even under these limitations, and we sketch ideas about how to address the limitations in the future. Furthermore, by designing this system hand-in-hand with gradual typing, we are able to execute even untyped code efficiently despite our reliance on nominal typing.

To support our claims about efficiency, we built a prototype compiler for a nominal object-oriented language and used it to implement key examples presented by Takikawa et al. Given that this involves a major shift in programming paradigms, we engineered the examples to exhibit the same complexity in terms of transitions between untyped and typed code as they did in Racket. Whereas Takikawa et al. measured overheads over 10,000% relative to the performance of fully untyped code, we measured worst-case overheads of less than 10%.

In summary, the contributions of this paper are as follows:

- We present new desirable properties of sound gradual type systems that we believe significantly improve their performance (Section 3).
- We present a simple gradually typed nominal object-oriented language (Section 5 through 7) that fulfills the properties traditionally desired of gradual type systems in addition to our own new properties (Section 8). We also give a crisp connection between the direct semantics of the language (Section 6) and the cast semantics of the language (Section 7).
- We provide evidence of our approach's feasibility and efficiency by presenting an implementation of said language and comparing benchmarks between it, Typed Racket [Takikawa et al. 2016], C# [Bierman et al. 2010], and Reticulated Python [Vitousek et al. 2014] (Section 9).
- We illustrate the significant tradeoffs and future challenges of our approach and sketch possible avenues for addressing its current limitations in the future (Section 10).

This work draws heavily from earlier work on gradual typing, which we review next.

2 BACKGROUND

Gradual typing, as originally proposed by Siek and Taha [2006], features two core elements: a special type **dyn** and a consistency relation $\tau \sim \tau'$ expressing that τ and τ' are structurally equal except for places featuring **dyn**. For example, the following types are consistent:

$$\textbf{dyn} \sim (\textbf{dyn} \to \textbf{dyn}) \sim (\textbf{int} \to \textbf{dyn}) \sim (\textbf{int} \to \textbf{bool})$$

A program in their gradually typed language type-checks according to the original typing rules, but with type equality replaced with the consistency relation in many places. This enables **dyn** to stand for any type while also maintaining the familiar static typing rules where **dyn** is not present. The gradually typed program is then translated into a variant of the original statically typed language by inserting dynamic casts where run-time checks are necessary to monitor the boundary between untyped and typed code.

Here is how this works for an example program:

$$f : \textbf{dyn} \to \textbf{dyn} \vdash (\lambda f' : \textbf{int} \to \textbf{int}.\ f'\ 5)\ f$$

The lambda term expects a parameter of type **int** → **int**, but it is applied to an argument of type **dyn** → **dyn**. Despite this difference in types, the program type-checks because these two types are considered to be consistent with each other. This gradually typed program is then translated to a statically typed program by inserting run-time casts, resulting in

$$f : \mathbf{dyn} \to \mathbf{dyn} \vdash (\lambda f' : \mathbf{int} \to \mathbf{int}.\ f'\ 5)\ (f :: \mathbf{int} \to \mathbf{int})$$

Here $e :: \tau$ represents a run-time check that e has type τ, conceptually throwing a run-time exception if it does not.

2.1 Casting Strategies

In most gradual typing settings, casts are the only source of run-time overhead incurred by gradual typing. Thus, where casts are inserted and how they work has a big impact on the performance of a gradually typed program. Before we suggest our own variation on casts in Section 3, we give an overview of existing casting strategies that have been studied as such.

2.1.1 Guarded. Most work on sound gradual typing—including the original works by Siek and Taha [2006], Tobin-Hochstadt and Felleisen [2006], Matthews and Findler [2007], and Gronski et al. [2006]—uses the *guarded* cast semantics. In those systems, a cast like the one above reduces as follows:

$$f :: \mathbf{int} \to \mathbf{int} \quad \mapsto \quad \lambda x : \mathbf{int}.\ (f\ (x :: \mathbf{dyn})) :: \mathbf{int}$$

Instead of checking whether the function f always returns an **int** when given an **int**, which is generally impossible, it is wrapped in a new function that upcasts its input to **dyn**—which always works—and, after the call to f completes, checks that its output is an **int**. Wadler and Findler [2009], and later Ahmed et al. [2011], showed that this is sound even if it is later discovered that f does not always return an **int** when given an **int**. However, instead of having one check at the point where the function is passed to the typed part of the program, this strategy will incur checks every time the function is called, which can cause signficant overhead if that function is heavily used. Simply wrapping functions into other functions also does not preserve object identity, which can be a problem in languages where object identity is semantically significant.

2.1.2 Transient. The *transient* cast semantics was proposed by Vitousek et al. [2014] to preserve object identity in Reticulated Python. It puts casts nearly everywhere in the code: the caller of a function casts an argument to the type that the function expects, but since a different caller might see that function as **dyn** → τ, the function itself also casts its parameters, leading to many unnecessary checks even in fully typed code. As such, soundness was not originally meant to be monitored in production programs, but rather intended to help with finding the sources of type errors during debugging. However, Vitousek et al. [2017] recently used this casting strategy as the basis of their work on open-world soundness, finding overheads much smaller than those reported by Takikawa et al. [2016], but still several multiples of the original run times, sometimes over 10x.

2.1.3 Monotonic. Another approach used by Vitousek et al. [2014] in Reticulated Python, and by Swamy et al. [2014] and Rastogi et al. [2015] in Safe TypeScript, is what Siek et al. [2015b] formalized as the *monotonic* approach. Here, every value keeps track of what type it has been checked to have, and enforces that type in later mutations. For example, the record $\{x : 5,\ y : \text{“}Hello''\}$ might be checked to have type $\{x : \mathbf{int}\}$, after which it will get a special run-time-type-information field assigned and become $\{x : 5,\ y : \text{“}Hello'',\ rtti : \{x : \mathbf{int}\}\}$. Any subsequent assignment of a non-integer value to x would fail, and future checks can use the information in *rtti* to fast-track failure or success instead of checking the value of x itself. The *rtti* field itself can only change monotonically towards more precise types (if they are consistent with the current values in the

structure). Applying this scheme to higher-order types is not straightforward; thus Swamy et al. do not treat a function **dyn** → **dyn** as compatible with **int** → **int**, while Siek et al. fall back to guarded semantics for function types.

2.2 Properties of Gradual Type Systems

Beyond soundness, there are additional desired properties for gradual type systems suggested by the literature. In the following, we describe what they are and why they are useful. Later, in Section 3, we propose two more properties related specifically to the efficiency of gradual typing.

2.2.1 Blame and Accountability. Lacking a proper word for a notion of "can assign blame correctly" as defined by Tobin-Hochstadt and Felleisen [2006] and Wadler and Findler [2009], we define *accountability* as the property that, when an inserted cast fails, it can refer the programmer to some untyped part of the program that is at fault. Higher-order types are what make blame hard to implement, since a higher-order cast cannot be determined right or wrong until later in the program when the cast function is supplied an argument. *Blame tracking* is the technique used to enable dynically created casts to keep track of the statically inserted cast they originated from.

2.2.2 The Gradual Guarantee. Siek et al. [2015a] defined the *gradual guarantee*, which expresses the idea that adding or removing type information from a program should not change its behavior in unexpected ways. In particular, making a well-typed program more dynamic should always result in a well-typed program that produces the same output. The only exception is that a more dynamic program can succeed where the original would fail because the original might assert some unnecessary and overly restrictive type cast.

The gradual guarantee thus captures the expectation that adding type annotations to an untyped program should preserve the semantics of the program *provided* those annotations are correct. While this clearly seems like a desirable property for gradually typed languages, Siek et al. [2015a] demonstrate that several existing gradual type systems do not satisfy this property, including Safe TypeScript [Swamy et al. 2014]. They remark that it seems "challenging to satisfy the gradual guarantee and efficiency at the same time".

2.3 Overhead of Gradual Typing

Recently, Takikawa et al. [2016] surveyed the state of performance evaluations on gradual type systems. They found that no gradually typed language had a systematic evaluation of the behavior of the language during the process of gradually typed software development, by which they mean an evaluation of how having mixed typed and untyped code affects run-time overheads. What they found instead was that if there was some kind of overhead evaluation, it usually just compared completely typed and completely untyped versions of programs. Thus, Takikawa et al. proposed a scheme of using microbenchmarks divided up into smaller modules. Each of these modules would exist in two versions, one completely typed, and one completely untyped. Thus, if a program consists of N modules, it would have 2^N potential configurations (i.e. different combinations of typed/untyped versions of the modules). Takikawa et al. created a suite of microbenchmarks in Typed Racket, and measured the overhead of gradual typing by comparing the running time of each configuration to the running time of the completely untyped configuration. While the completely typed configuration was usually about 30% faster than the completely untyped one, they found some programs had configurations with over 10,000% overhead. Furthermore, for some programs there was no sequence of annotating modules (simulating a gradual evolution from a completely untyped to a completely typed program) where every intermediate configuration had less than 1,000% overhead. We agree with their assessment that such overheads are far beyond acceptable, and that sound gradual typing needs significant improvements in efficiency in order to be adopted.

2.4 Gradual Typing for Object-Oriented Languages

Gradual typing was extended to object-oriented languages quite early, again by Siek and Taha [2007]. Their approach was based on structural subtyping on records. They used the guarded casting strategy, even delaying checks for the presence of expected fields to whenever that field was actually accessed. This is an example of how design choices in casting strategies are not limited to just functions. Since the language we are formalizing and have implemented is a nominal object-oriented language, it touches on many aspects from prior work on sound gradual object-oriented languages (both nominal and structural). Such languages include C# [Bierman et al. 2010], GradualTalk [Allende et al. 2014], Reticulated Python [Vitousek et al. 2014], Safe TypeScript [Rastogi et al. 2015; Swamy et al. 2014], and StrongScript [Richards et al. 2015]. We discuss their relations to our work as we get to the relevant parts of the paper.

3 TOWARDS WELL-BEHAVED AND EFFICIENT GRADUAL TYPING

In the light of the previous discussion, we want to devise a sound gradually typed language that is accountable, fulfills the gradual guarantee, and has acceptably low overhead for the checks needed to ensure soundness. Since the overhead of gradual typing comes from the run-time checks it needs to insert, we aim to minimize the number and cost of those checks. The main ingredients of our scheme to achieve this goal are nominality and run-time type information. The idea is that every value will be tagged with its most precise type as run-time type information. This enables what we call *transparency* and *immediate* accountability, the combination of which provides efficiency.

In this section, we give a brief overview of what these ingredients are and how our approach relates to existing work. We formalize transparency and immediate accountability in Section 8.

3.1 Transparency

A transparent casting strategy is one in which a cast is invisible to the runtime system after it is evaluated, unless of course it fails. Thus, guarded casting is not transparent because a cast can wrap a value with a new value that would otherwise not be present. Transient casting, on the other hand, is transparent because the value is simply passed on after the cast succeeds. Monotonic casting provides a middle ground in which the same value is passed on, but the value is modified in place.

3.2 Immediate Accountability

Accountability is the ability to identify a source of a cast failure in the source program. *Immediate* accountability is the ability to identify that source immediately as it is being executed. In other words, loops and recursion aside, once execution has successfully proceeded past a point in the program, then that point cannot be at fault for some future cast failure. None of guarded casting, transient casting, or monotonic casting are necessarily immediately accountable. They often only do shallow aspects of a cast immediately, and defer deep aspects of a cast to later. C# [Bierman et al. 2010] and Safe TypeScript [Swamy et al. 2014] are the only prior gradual type systems that we know of that are immediately accountable, both of which sacrifice the gradual guarantee to achieve this.

3.3 Run-Time Type Information

Having every value always be tagged with its most precise type requires a significant assumption: every value's most-precise type must be known upon construction of the value, even if it is constructed in an untyped part of the program. We discuss the implications of this requirement next and in Section 10.3.

3.4 Discussion

Most earlier work on gradually typing focuses on adding gradual typing to an existing system. Half of this work aims to add gradual typing to an existing untyped language. Examples of this category are Reticulated Python [Vitousek et al. 2014], Gradualtalk [Allende et al. 2014], Safe TypeScript [Rastogi et al. 2015; Swamy et al. 2014], and StrongScript [Richards et al. 2015], as well as the two widespread unsound gradually typed languages (preferably referred to as *optionally* typed languages [Bracha 2004]), Hack [Facebook, Inc. 2016] and TypeScript [Microsoft 2012]. The other half aims to add gradual typing to, i.e. "gradualize", an existing typed language. Examples of this category are C# [Bierman et al. 2010], gradual typing for generics by Ina and Igarashi [2011], and work on systematically [Garcia et al. 2016] or automatically [Cimini and Siek 2016] gradualizing given typed languages.

Certainly much of the appeal of gradual typing is that it can give a preexisting language new access to the counterpointing paradigm. However, both directions currently have weaknesses to overcome due to the fact that gradual typing is heavily intertwined with both the type system and the runtime implementation. Adding sound gradual typing to an untyped language seems to frequently incur significant overhead, sometimes making programs multiple orders of magnitude slower [Takikawa et al. 2016]. Part of the problem is that the type-system features needed to capture the idioms common to untyped languages are not easy to check efficiently, especially when the underlying runtime is not designed for it.

Conversely, adding gradual typing to a typed language can introduce unexpected behavior due to violations of the gradual guarantee. For example, in C#, adding more precise type information to a well-typed program may cause that program to cease being well-typed, as the new information may introduce ambiguities (e.g. through additional available overloadings) that would have to be resolved. When such an ambiguity is introduced at compile time, C# can rely on the programmer to resolve the error. However, with gradual typing, such ambiguities can be introduced at run time, where no such programmer is readily available to resolve the problem, causing the system to throw a run-time error. Furthermore, C# compilation is heavily type-directed, but gradual typing often makes type information available only at run time, so C# is forced to defer much of its compilation of untyped code to run time. We have found that this can introduce significant overhead, as we illustrate in Section 9. We discuss these and other issues in more detail in Section 10. The main point here is that gradual typing is not easy to bolt onto existing languages without serious drawbacks.

Thus, in contrast to most earlier work, we focus on gradual typing for new systems, where the entire language can be designed from the start to both support and benefit from gradual typing. Clearly we can benefit from all the work on adding gradual typing to existing systems, but our change in focus also enables us to benefit from a greater degree of flexibility. Here we use that flexibility to address the efficiency issues in prior work while retaining desirable properties such as accountability and the gradual guarantee. While the improvement in performance is certainly more noticeable when compared to systems that have added sound gradual typing to untyped languages, we even achieve better performance than systems that have added sound gradual typing to typed languages. We accomplish this by designing a language with a nominal runtime environment, which is where most of our performance gains come from, optimized for gradual typing, which is where our smaller performance gains come from. Nominality in and of itself is not a guarantee for good performance, nor does it imply transparency or accountability. For example, our benchmarks for C#—which is nominal, transparent, and immediately accountable—show that its dynamically typed parts are quite slow (see Section 9). As another example, StrongScript [Richards et al. 2015] uses nominality for performance in fully typed programs, but the language as a whole is neither transparent nor immediately accountable, and there is no performance evaluation of mixed

programs, where Takikawa et al. [2016] found the biggest problems. Furthermore, Richards et al. found that blame tracking produced significant overhead, prompting them to only evaluate the performance of their system without blame.

Of course, the nominality of our runtime environment restricts the programmer. While gradual typing can recover some of the expressiveness of structural typing that prior research has worked hard to preserve, there is still much that is lost. We expect to address this by developing methods for mixing structural values into our nominal system, much like we mix untyped and typed code. And in fact, there is already significant work to this effect, some with [Richards et al. 2015; Wrigstad et al. 2010] and some without gradual typing [Anderson and Drossopoulou 2003]. But it is important to recognize that adding structural reasoning is not necessary for many of the well-known applications of gradual typing. Much like how we focus on gradual typing being a part of the language from the beginning, one envisions gradual typing being a part of the software-development process from the beginning. Stable code would typically be typed, benefiting from better optimization and providing machine-checkable document for programmers and IDEs interacting with this code. Meanwhile unstable code would not need to be typed, which is useful for prototyping, scripting, or simply letting the programmer first experiment in the paradigm they are most comfortable with. In particular, student programmers can enjoy the benefits of working with well-typed APIs without having the type system impede their first explorations into programming.

What we present in this paper is a minimal system striving towards this end, just large enough to test whether this path has promise. Our formalization presented in Section 5 is sufficient for covering the same feature set as Featherweight Java [Igarashi et al. 2001] with interfaces and little more. Meanwhile, we have made an effort to be forwards compatible with a multitude of features frequently found in nominal industry languages, all while also making an effort to be forwards compatible with structural values. Our implementation covers a much larger subset of pre-generics Java, including assignment, interfaces, overloading, primitive types, messages to super, access control, and null pointers. Some of these features were adapted to work with gradual typing in a way that satisfies the gradual guarantee. For example, we require that all overloadings of a method be disjoint in order to avoid ambiguities at dynamic method lookup at run time, and we made null explicit in anticipation of adding generics types later to avoid problems with both type-argument inference [Smith and Cartwright 2008] and unsoundness [Amin and Tate 2016]. While we have yet to decide how to accommodate structural values, we will discuss a number of possible strategies to this end in Section 10.3.

4 THE OPTIMISTIC PERSPECTIVE

Throughout the remainder of this paper we will be using the terms *optimistic* and *pessimistic*. This is a change in terminology that we find unifies our definitions. The idea is that there are two attitudes towards typing. One is the optimistic attitude: programs should be able to proceed so long as they might succeed. Dynamic typing takes this attitude, trying to only stop a program when execution encounters an issue that cannot be overcome. The other is the pessimistic attitude: programs should only be able to proceed when it is known they will succeed. Static typing takes this attitude, trying to only compile a program if it is exhibits certain guarantees.

Both attitudes have their advantages and disadvantages, and consequently are each better suited to different circumstances. The purpose of gradual typing is to give the programmer the ability to explicitly control which attitude is applied where in a given program. Thus, when a variable is given the type **`dynamic`**, the programmer is directing the compiler to treat the variable as optimistically having whatever type is necessary for the usages at hand. On the other hand, when a variable is given the type **Number**, the programmer is directing the compiler to treat the variable as

Class/Interface Name C Field Name f Method Name m Variable Name x

$$
\begin{array}{rrl}
\text{Type} & \tau & ::= \top \mid C \mid \textbf{dynamic} \\
\text{Context} & \Gamma & ::= \cdot \mid \Gamma, \tau\ x \\
\text{Expression} & e & ::= x \mid \textbf{let}\ \tau\ x := e\ \textbf{in}\ e \mid C(e, \ldots) \mid e.f_\delta \mid e.m_\delta(e, \ldots) \mid \textbf{cast}\ e\ \textbf{to}\ \tau \\
\text{Dispatch Mode} & \delta & ::= C \mid \textbf{dynamic} \\
\text{Method Signature} & s & ::= \tau\ m(\Gamma) \\
\text{Method Definition} & d & ::= s \mapsto e \\
\text{Interface Definition} & i & ::= \textbf{interface}\ C\ \{s; \ldots\} \\
\text{Class Definition} & c & ::= \textbf{class}\ C(\tau\ f, \ldots)\ \textbf{implements}\ C, \ldots\ \{d; \ldots\} \\
\text{Environment Definition} & \Psi & ::= \cdot \mid \Psi, i \mid \Psi, c
\end{array}
$$

Fig. 1. Grammar

pessimistically only being usable where a **Number** provides sufficient guarantees. In this way, a gradually typed language enables the programmer to change attitudes as they see fit.

In technical terms, this new terminology expresses the same concept that Garcia et al. [2016] generalize as *consistent lifting*.

5 THE TYPE SYSTEM

The grammar of our small gradually typed object-oriented language is shown in Figure 1. The grammar is mostly standard besides being fairly minimal. Our implementation does of course handle a much richer set of features as described in Section 9. The point of this formalization is not to specify our implemented language, but to be able to discuss the interesting aspects of our approach: nominality, run-time type information, transparency, and immediate accountability. Note that as another simplification, we do not concern ourselves with naming issues; it is obvious how to adjust the rules throughout this paper to address problems such as name shadowing.

5.1 Dispatch Modes

The one irregular feature of our grammar is the use of *dispatch modes* δ. Every field access and method invocation is annotated with a dispatch mode. This reflects the fact that, at compile time, one must decide how a field should be accessed or a method implementation should be looked up. For example, a method could be looked up by accessing some offset of the object's virtual-method table. In this case, the dispatch mode is the class that specifies which offset to use. Alternatively, a method could be looked up by searching through the object's interface table, in which case the dispatch mode is the interface to search for. Lastly, since we are providing a gradually typed language, a method could be looked up in the object's hashtable, like one would do in a dynamically-typed language such as Python. In this case, the dispatch mode is **dynamic**.

Note that this means we view objects as supplying both a virtual-method table[1] (and interface table) *and* a (possibly immutable and shared) hashtable. Similarly, fields can be accessed through fixed offsets when the object's class is known, or through the hashtable when the field access is being typed dynamically. This allows us to interact with objects efficiently regardless of the typing attitude we happen to be applying in a given part of the program. Although in theory we could develop a more traditional calculus without dispatch modes, we include them here to better illustrate how we are able to implement gradual typing.

[1]For simplicity, we do not allow classes to extend other classes. However, we have designed our calculus to support class inheritance, and our implementation supports it as well.

$$\frac{}{\Psi \vdash C\ S\ C} \qquad \frac{C\ \textbf{implements}\ C' \in \Psi}{\Psi \vdash C\ S\ C'} \qquad \frac{}{\Psi \vdash \tau\ S\ \top} \qquad \frac{}{\Psi \vdash \tau\ S\ \textbf{dynamic}} \qquad \frac{}{\Psi \vdash \textbf{dynamic} \triangleleft C}$$

Fig. 2. Subtyping, where S is optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$

$$\boxed{\frac{\Psi \vdash \tau \qquad \Psi \vdash e\ S\ \tau}{\Psi \vdash_S e} \qquad \frac{\Psi \mid \cdot \vdash e\ S\ \tau}{\Psi \vdash e\ S\ \tau}} \qquad \frac{\tau\ x \in \Gamma \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash x\ S\ \tau'} \qquad \frac{\Psi \vdash \tau \qquad \Psi \mid \Gamma \vdash e\ S\ \tau \qquad \Psi \mid \Gamma, \tau\ x \vdash e'\ S\ \tau'}{\Psi \mid \Gamma \vdash \textbf{let}\ \tau\ x := e\ \textbf{in}\ e'\ S\ \tau'}$$

$$\frac{\textbf{class}\ C(\tau_1, \ldots, \tau_n) \in \Psi \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i\ S\ \tau_i \qquad \Psi \vdash C\ S\ \tau}{\Psi \mid \Gamma \vdash C(e_1, \ldots, e_n)\ S\ \tau} \qquad \frac{\Psi \vdash \delta.f : \tau \qquad \Psi \vdash e\ S\ \delta \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash e.f_\delta\ S\ \tau'}$$

$$\frac{\Psi \vdash \delta.m(\tau_1, \ldots, \tau_n) : \tau \qquad \Psi \mid \Gamma \vdash e\ S\ \delta \qquad \forall i.\ \Psi \mid \Gamma \vdash e_i\ S\ \tau_i \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash e.m_\delta(e_1, \ldots, e_n)\ S\ \tau'} \qquad \frac{\Psi \mid \Gamma \vdash e\ S\ \top \qquad \Psi \vdash \tau\ S\ \tau'}{\Psi \mid \Gamma \vdash \textbf{cast}\ e\ \textbf{to}\ \tau\ S\ \tau'}$$

$$\frac{\textbf{class}\ C(\tau_1\ f_1, \ldots) \in \Psi}{\Psi \vdash C.f_i : \tau_i} \qquad \frac{\tau\ C.m(\tau_1, \ldots, \tau_n) \in \Psi}{\Psi \vdash C.m(\tau_1, \ldots, \tau_n) : \tau}$$

$$\frac{}{\Psi \vdash \textbf{dynamic}.f : \textbf{dynamic}} \qquad \frac{}{\Psi \vdash \textbf{dynamic}.m(\textbf{dynamic}, \ldots) : \textbf{dynamic}}$$

$$\frac{}{\Psi \vdash \top} \qquad \frac{\textbf{interface}\ C \in \Psi}{\Psi \vdash C} \qquad \frac{\textbf{class}\ C \in \Psi}{\Psi \vdash C} \qquad \frac{}{\Psi \vdash \textbf{dynamic}} \qquad \frac{}{\Psi \vdash \cdot} \qquad \frac{\Psi \vdash \Gamma \qquad \Psi \vdash \tau}{\Psi \vdash \Gamma, \tau\ x}$$

Fig. 3. Expression Typing, where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping

In general, the dispatch modes will be inferred by the compiler. As this issue is orthogonal to the properties that we are trying to formalize and comes with its own interesting design choices, we defer discussion of dispatch-mode inference to the supplementary material.

5.2 Subtyping

Our system provides two kinds of subtyping: optimistic and pessimistic. Optimistic subtyping ($\triangleleft$) recognizes that **dynamic** is optimistically a subtype of any type τ because it can optimistically be interpreted as being τ. Pessimistic subtyping ($\blacktriangleleft$) ensures that one type is a subtype of another only if all values of the former type are also values of the latter type. The two differ by only rule, so we use the metavariable S to formalize both of them simultaneously in Figure 2.

Like most subtyping relations, pessimistic subtyping is transitive. However, optimistic subtyping, like its inspiration *consistent* subtyping [Siek and Taha 2007], is *not* transitive because it conceptually confuses existentials with universals. That is, **dynamic** semantically represents $\exists\alpha.\alpha$. Consequently, every type is semantically a subtype of **dynamic**, as is captured by both pessimistic and optimistic subtyping. But the optimistic attitude says to also treat **dynamic** as $\forall\alpha.\alpha$ *when it would make the subtyping hold*, making **dynamic** an optimistic subtype of every type. Thus the difference between

$$\frac{\Psi \vdash \Psi}{\vdash \Psi} \qquad \frac{}{\Psi \vdash \cdot} \qquad \frac{\Psi \vdash \Psi' \qquad \Psi \vdash i}{\Psi \vdash \Psi', i} \qquad \frac{\Psi \vdash \Psi' \qquad \Psi \vdash c}{\Psi \vdash \Psi', c}$$

$$\frac{\forall i.\ \Psi \vdash s_i}{\Psi \vdash \textbf{interface}\ C\ \{s_1; \dots\}} \qquad \frac{\begin{array}{c}\forall i.\ \Psi \vdash \tau_i \qquad \forall i.\ \Psi \mid C \vdash d_i \\ \forall i.\ \textbf{interface}\ C_i\ \{s_1^i; \dots\} \in \Psi \qquad \forall i. \forall j. \exists k_{i,j}.\ \Psi \vdash d_{k_{i,j}} \triangleleft s_j^i\end{array}}{\Psi \vdash \textbf{class}\ C(\tau_1\ f_1, \dots)\ \textbf{implements}\ C_1, \dots\ \{d_1; \dots\}}$$

$$\frac{\Psi \vdash \tau \qquad \Psi \vdash \Gamma}{\Psi \vdash \tau\ m(\Gamma)} \qquad \frac{\Psi \vdash \tau \qquad \Psi \vdash \Gamma \qquad \Psi \mid C\ \textbf{this}, \Gamma \vdash e \triangleleft \tau}{\Psi \mid C \vdash \tau\ m(\Gamma) \mapsto e}$$

$$\frac{}{\Psi \vdash \cdot \triangleleft \cdot} \qquad \frac{\Psi \vdash \Gamma \triangleleft \Gamma' \qquad \Psi \vdash \tau \triangleleft \tau'}{\Psi \vdash \Gamma, \tau\ x \triangleleft \Gamma, \tau'\ x} \qquad \frac{\Psi \vdash \Gamma' \triangleleft \Gamma \qquad \Psi \vdash \tau \triangleleft \tau'}{\Psi \vdash \tau\ m(\Gamma) \mapsto e \triangleleft \tau'\ m(\Gamma')}$$

Fig. 4. Class and Interface Validation

optimistic and pessimistic subtyping captures the difference between the optimistic and pessimistic attitudes.

5.3 Expression Typing

Our expression-typing rules are shown in Figure 3. Observe that they look nearly identical to what one might expect for a statically typed language. The only other major difference is that they are parameterized by a subtyping relation S. When one uses optimistic subtyping $\triangleleft$ for S, we say the expression type-checks optimistically. Likewise, when one uses pessimistic subtyping $\blacktriangleleft$ for S, we say the expression type-checks pessimistically. This parameterization illustrates that type-checking is both standard and adjustable to the preferred attitude at hand.

5.4 Class and Interface Validation

Class and interface validation is shown in Figure 4. Once again it is quite standard. The one point to note is that a class is allowed to only optimistically satisfy method signatures of implemented interfaces. In this way the class implementation can be completely untyped, even if it is implementing typed interfaces. The only requirement then is that the class specify the list of interfaces it intends to implement, and at least provide methods with the appropriate names and arities. Note also that method definitions are always type-checked optimistically. Consequently, one of the challenges is to achieve sound gradual typing throughout the class hierarchy.

6 THE DIRECT SEMANTICS

Traditionally, sound gradually typed calculi are formalized using a type-directed translation to a cast calculus [Cimini and Siek 2016; Henglein 1994; Siek and Taha 2007, 2006]. We will do so as well in the next section, but here we first develop an operational semantics directly on our calculus. The intent is to provide an intuitive semantics that programmers can use to reason about how their gradually typed programs will behave without needing to understand the details of when and where casts are inserted and how they are implemented. In the next section, we will demonstrate that there is a strong relationship between these direct semantics and the ones derived from cast insertions.

We formalize the direct semantics of our calculus using rewrite rules, as presented in Figure 5. This formalization is odd in that some of the assumptions of the various rules are parenthesized.

Grammar

$$
\begin{array}{rrl}
\text{Value} & v & ::= C(v, \dots) \\
\text{Error} & \varepsilon & ::= v.f_{\mathtt{dynamic}} \mid v.m_{\mathtt{dynamic}}(v, \dots) \mid \mathbf{cast}\ v\ \mathbf{to}\ C \\
\text{Valuation} & \nu & ::= v \mid \varepsilon \mid \infty \\
\text{Evaluation Context} & E & ::= \cdot \mid \mathbf{let}\ \tau\ x := E\ \mathbf{in}\ e \mid C(v, \dots, E, e, \dots) \mid E.f_\delta \\
& & \quad \mid E.m_\delta(e, \dots) \mid v.m_\delta(v, \dots, E, e, \dots) \mid \mathbf{cast}\ E\ \mathbf{to}\ \tau \\
\text{Method Implementation} & \bar{d} & ::= \tau\ m_\delta(\Gamma) \mapsto e \\
\text{Class Implementation} & \bar{c} & ::= \mathbf{class}\ C(\tau\ f, \dots)\ \mathbf{implements}\ C, \dots\ \{\bar{d}; \dots\} \\
\text{Environment Implementation} & \bar{\Psi} & ::= \cdot \mid \bar{\Psi}, i \mid \bar{\Psi}, \bar{c}
\end{array}
$$

Terminals

$$\frac{\Psi \vdash v \blacktriangleleft \tau}{\Psi \vdash v\ \mathbf{terminal}\ \tau} \qquad \frac{\Psi \vdash e\ \mathbf{erroneous}}{\Psi \vdash e\ \mathbf{terminal}\ \tau} \qquad \frac{\Psi \vdash e\ \mathbf{bad\text{-}cast}}{\Psi \vdash e\ \mathbf{erroneous}}$$

$$\frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad v = C(\dots) \quad \neg\, \Psi \vdash C \blacktriangleleft C'}{\Psi \vdash E[\mathbf{cast}\ v\ \mathbf{to}\ C']\ \mathbf{bad\text{-}cast}} \qquad \frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad v = C(\dots) \quad \mathbf{class}\ C(f^1, \dots) \in \Psi \quad \nexists i.\ f = f^i}{\Psi \vdash E[v.f_{\mathtt{dynamic}}]\ \mathbf{erroneous}}$$

$$\frac{\nexists e'.\ \Psi \vdash e \twoheadrightarrow e' \quad \neg\, \Psi \vdash e\ \mathbf{terminal}\ \tau}{\Psi \vdash e\ \mathbf{lapse}\ \tau} \qquad \frac{\Psi \vdash_\blacktriangleleft E \quad \Psi \vdash_\blacktriangleleft v \quad \forall i.\ \Psi \vdash_\blacktriangleleft v_i \quad v = C(\dots) \quad \nexists \tau, \tau_1, \dots, \tau_n.\ \tau\ C.m(\tau_1, \dots, \tau_n) \in \Psi}{\Psi \vdash E[v.m_{\mathtt{dynamic}}(v_1, \dots, v_n)]\ \mathbf{erroneous}}$$

Valuations

$$\frac{\bar{\Psi} \vdash e\ R^*\ v \quad \bar{\Psi} \vdash v \blacktriangleleft \tau}{\bar{\Psi} \vdash e\ R^\infty\ v : \tau} \qquad \frac{\bar{\Psi} \vdash e\ R^*\ E[\varepsilon] \quad \bar{\Psi} \vdash E[\varepsilon]\ \mathbf{erroneous}}{\bar{\Psi} \vdash e\ R^\infty\ \varepsilon : \tau} \qquad \frac{\bar{\Psi} \vdash e\ R^\infty}{\bar{\Psi} \vdash e\ R^\infty\ \infty : \tau} \qquad \frac{\Psi \vdash e\ R^*\ e' \quad \Psi \vdash e'\ \mathbf{lapse}\ \tau}{\Psi \vdash e\ R^*\ \mathbf{lapse}\ \tau}$$

Evaluation Contexts

$$\frac{}{\Psi \vdash_S \cdot} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S \mathbf{let}\ \tau\ x := E\ \mathbf{in}\ e} \qquad \frac{\mathbf{class}\ C(\tau_1, \dots, \tau_n) \in \Psi \quad \forall j.\ \Psi \vdash v_j\ S\ \tau_j \quad \Psi \vdash_S E}{\Psi \vdash_S C(v_1, \dots, v_i, E, e_{i+2}, \dots, e_n)} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S E.f_\delta}$$

$$\frac{\Psi \vdash_S E}{\Psi \vdash_S E.m_\delta(e_1, \dots)} \qquad \frac{\Psi \vdash \delta.m(\tau_1, \dots, \tau_n) : \tau \quad \Psi \vdash v\ S\ \delta \quad \forall j.\ \Psi \vdash v_j\ S\ \tau_j \quad \Psi \vdash_S E}{\Psi \vdash_S v.m_\delta(v_1, \dots, v_i, E, e_{i+2}, \dots, e_n)} \qquad \frac{\Psi \vdash_S E}{\Psi \vdash_S \mathbf{cast}\ E\ \mathbf{to}\ \tau}$$

Reductions

$$\frac{\bar{\Psi} \vdash e\ R\ e' \quad (\bar{\Psi} \vdash_\triangleleft E)}{\bar{\Psi} \vdash E[e]\ R\ E[e']} \qquad \frac{(\bar{\Psi} \vdash v \triangleleft \tau)}{\bar{\Psi} \vdash \mathbf{let}\ \tau\ x := v\ \mathbf{in}\ e\ R\ e[x \mapsto v]} \qquad \frac{v = C(v_1, \dots) \quad \mathbf{class}\ C(f^1, \dots) \in \bar{\Psi} \quad (\bar{\Psi} \vdash v \triangleleft \delta)}{\bar{\Psi} \vdash v.f^i_\delta\ R\ v_i}$$

$$\frac{v = C(\dots) \quad C.m_\delta(\tau_1\ x_1, \dots, \tau_n\ x_n) \mapsto e \in \bar{\Psi} \quad (\bar{\Psi} \vdash v \triangleleft \delta) \quad (\forall i.\ \bar{\Psi} \vdash v_i \triangleleft \tau_i)}{\bar{\Psi} \vdash v.m_\delta(v_1, \dots, v_n)\ R\ e[\mathbf{this} \mapsto v, x_1 \mapsto v_1, \dots, x_n \mapsto v_n]} \qquad \frac{v = C(\dots) \quad \bar{\Psi} \vdash C \blacktriangleleft \tau \quad (\bar{\Psi} \vdash v \triangleleft \top)}{\bar{\Psi} \vdash \mathbf{cast}\ v\ \mathbf{to}\ \tau\ R\ v}$$

Fig. 5. Operational Semantics, where R is either optimistic $\twoheadrightarrow$ (ignoring parenthesized assumptions) or pessimistic $\rightarrow$ (asserting parenthesized assumptions) reduction

This is because the rules are parameterized by a reduction relation R that can stand for either *optimistic reduction* ($\rightarrowtriangle$) or *pessimistic reduction* ($\rightarrow$). For optimistic reduction, one ignores the parenthesized assumptions, optimistically hoping that the expected invariants of the system hold. For pessimistic reduction, one includes the parenthesized assumptions, pessimistically asserting the expected invariants of the system throughout execution. Obviously pessimistic reduction provides more guarantees, but optimistic reduction is much more efficient. Thus we can gain much from understanding the relationship between these two semantics.

Values, Valuations, and Lapses. The values in our system are instances of classes. The arguments to the class constructor indicate the object's values for the class's fields.

Note that **`error`** is not an expression in our formalization. Instead, we simply let failing casts get stuck. This means even non-value programs can get stuck for both acceptable and unacceptable reasons. For example, a program could be stuck because it is a failed cast, which is acceptable and would be caught by the runtime system. However, a program could also be stuck because it is trying to access a field at a memory offset not provided by the object, which is unacceptable and corresponds to a potentially dangerous memory-access violation. We use the judgement $\Psi \vdash e$ **`terminal`** τ, defined in Figure 5, to indicate when e is stuck for an acceptable reason with type τ. In particular, e could be a value of type τ, a failed dynamic field lookup, a failed dynamic method lookup, or a failed cast. As a convenience, we also use the counterpoint judgement $\Psi \vdash e$ **`lapse`** τ to indicate when e is *pessimistically* stuck for a reason unacceptable for type τ, which we call a lapse because it indicates a current violation of some intended invariant.

Each of these cases represents a different observable result of executing a program. We use *valuations* v to represent the acceptable results. The idea is that, ignoring situations where a program lapses, a program's semantics are the valuations it can result in. Since a program might fail to terminate, we include ∞ as a valuation representing when programs execute forever. We capture valuations with the judgement $\bar{\Psi} \vdash e\ R^{\infty}\ v : \tau$, defined in Figure 5. As a convenience, we also use the counterpoint judgement $\Psi \vdash e\ R^{*}$ **`lapse`** τ to indicate that e results in some unacceptable lapse rather than an acceptable valuation.

Reductions. Now we discuss the reduction rules in more detail. As we mentioned before, these rules specify both optimistic reduction ($\rightarrowtriangle$), which ignores the parenthesized assumptions, and pessimistic reduction ($\rightarrow$), which asserts the parenthesized assumptions. Pessimistic reduction of evaluation contexts uses the judgement $\Psi \vdash_{\triangleleft} E$ to ensure that evaluation of expressions only moves on from left to right when the already computed values actually have their expected types. The use of assertions aside, the reduction rules are standard except for one oddity in our semantics for method invocations.

In particular, the assumption $C.m_{\delta}(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \in \bar{\Psi}$ looks up class C's *implementation* for method *m and dispatch mode δ* in the environment *implementation* $\bar{\Psi}$. The most important detail of this assumption is the inclusion of the dispatch mode δ in this lookup. This allows class C to provide a different implementation of m for each appropriate dispatch mode. This will enable C to address the fact that its method definition only *optimistically* satisfies the signatures of the interfaces it implements. To understand how, let us consider implementations in more detail.

Implementations. Whereas our typing rules are defined in the context of an environment *definition*, our reduction rules are defined in the context of an environment *implementation*. The two differ in that the former specifies class definitions, whereas the latter specifies class implementations. A class definition provides a method definition for each method m of the class; a class implementation provides a method implementation for each method m of the class *and each suitable dispatch mode δ*

$$\frac{\Psi \mid \Psi \vdash_S \bar{\Psi}}{\Psi \vdash_S \bar{\Psi}} \qquad \frac{}{\Psi \mid \cdot \vdash_S \cdot} \qquad \frac{\Psi \mid \Psi' \vdash_S \bar{\Psi}}{\Psi \mid \Psi', i \vdash_S \bar{\Psi}, i} \qquad \frac{\Psi \mid \Psi' \vdash_S \bar{\Psi} \qquad \Psi \mid c \vdash_S \bar{c}}{\Psi \mid \Psi', c \vdash_S \bar{\Psi}, \bar{c}}$$

$$\frac{\begin{array}{c}\forall i.\exists j_i.\ \Psi \mid C \mid d_{j_i} \vdash_S \bar{d}_i \\ \forall i.\exists j_i.\ \Psi \vdash \bar{d}_{j_i} :_C d_i \qquad \forall i.\exists j_i.\ \Psi \vdash \bar{d}_{j_i} :_{\mathbf{dynamic}} d_i \\ \forall i.\ \mathbf{interface}\ C_i\ \{s_1^i; \ldots\} \in \Psi \qquad \forall i.\forall j.\exists k_{i,j}.\ \Psi \vdash \bar{d}_{k_{i,j}} :_{C_i} s_j^i\end{array}}{\Psi \mid \mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1; \ldots\} \vdash_S \mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{\bar{d}_1; \ldots\}}$$

$$\frac{\begin{array}{c}\forall i.\ \Psi \vdash \tau_i' \triangleleft \tau_i \qquad \Psi \vdash \tau \triangleleft \tau' \qquad \Psi \mid C\ \mathbf{this}, \tau_1'\ x_1, \ldots, \tau_n'\ x_n \vdash e'\ S\ \tau' \\ \Psi \vdash \mathbf{let}\ \tau_1\ x_1 \coloneqq x_1\ \mathbf{in}\ \ldots \mathbf{let}\ \tau_n\ x_n \coloneqq x_n\ \mathbf{in}\ \mathbf{let}\ \tau\ x \coloneqq e\ \mathbf{in}\ x \preceq e' : \tau'\end{array}}{\Psi \mid C \mid \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n) \mapsto e \vdash_S \tau'\ m_\delta(\tau_1'\ x_1, \ldots, \tau_n'\ x_n) \mapsto \mathbf{let}\ \tau'\ x \coloneqq e'\ \mathbf{in}\ x}$$

$$\frac{\Psi \vdash \bar{d} :_\delta s}{\Psi \vdash \bar{d} :_\delta s \mapsto e} \qquad \frac{}{\Psi \vdash \tau\ m_C(\Gamma) \mapsto e :_C \tau\ m(\Gamma)}$$

$$\frac{}{\Psi \vdash \mathbf{dynamic}\ m_{\mathbf{dynamic}}(\mathbf{dynamic}\ x_1, \ldots, \mathbf{dynamic}\ x_n) \mapsto e :_{\mathbf{dynamic}} \tau\ m(\tau_1\ x_1, \ldots, \tau_n\ x_n)}$$

Fig. 6. Implementation Validation, where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping

for m. The body of each such method implementation is a slightly adjusted version of the body of the method definition to account for the corresponding dispatch mode, as we will describe below.

We formalize implementations of definitions in Figure 6. The judgement $\Psi \vdash_S \bar{\Psi}$ indicates that $\bar{\Psi}$ is a valid implementation of the environment definition Ψ. Furthermore, if the parameter S is optimistic subtyping ($\triangleleft$), then the body of every method implementation in $\bar{\Psi}$ is optimistically typed. Likewise, if the parameter S is pessimistic subtyping ($\blacktriangleleft$), then the body of every method implementation in $\bar{\Psi}$ is pessimistically typed.

A class implementation $\bar{c}$ is valid for a class definition c if every method implementation in $\bar{c}$ corresponds to some method definition in c *and* every method definition in c has a corresponding method implementation in $\bar{c}$ for each necessary dispatch mode. In particular, there must be an implementation for the dispatch modes corresponding to the class itself and to **dynamic** dispatch. Furthermore, if a method definition is used to satisfy some method signature in an interface implemented by the class, then there must be an implementation for the dispatch mode corresponding to that interface. Thus, a class implementation simply specifies the contents of the virtual-method table, interface table, and dispatch hashtable, but with each way to dispatch a given method having its own implementation (employing low-level tricks to keep the size of the executable down).

Each of these method implementations corresponds to the same method definition, and while that implies they are closely related, it does not imply they are identical. First, the signature of a method implementation coincides with the signature corresponding to its own dispatch mode, *not* to its method definition. Second, the body of the method implementation needs to be adjusted to conform with the corresponding signature. For example, consider a method implementation whose dispatch mode is an interface implemented by the class. The body of the method definition is defined in terms of the class's signature for the method, but that signature only *optimistically* satisfies the signature of the method required by the interface.

We address this difference by inserting variable assignments to retype the method parameters and return value according to the method signature. Next, the *refinement* relation ($\preceq$) specifies that the actual method body e' of the implementation is a refinement of the original body wrapped in these retyping expressions, which means the implementation can have casts inserted to check

optimistic assumptions made in the method definition. Refinement is a relation, not a procedure, which means the refined expression may have no additional casts at all, or just the right amount to type-check pessimistically (in addition to optimistically), or many more than necessary. We defer detailed discussion of refinement until the next section.

Given an environment definition Ψ, there exists a naïve implementation of Ψ. In particular, because refinement is reflexive, one can simply define every method implementation to be the body of the corresponding method definition modulo retyping the inputs and output. As an abuse of notation, we refer to this naïve implementation as Ψ. If Ψ is a valid environment definition, then it is trivial to prove that Ψ is also a valid *optimistically*-typed implementation of itself.

Similarly, given an environment implementation $\bar{\Psi}$, there often exists a corresponding definition for $\bar{\Psi}$. In particular, one derives a class C's definition of a method from that method's implementation for the dispatch mode C. As an abuse of notation, we refer to this corresponding definition as $\bar{\Psi}$. If $\bar{\Psi}$ is a valid implementation of some valid environment definition Ψ, then the definition $\bar{\Psi}$ has exactly the same typing information as Ψ.

Soundness. Even without inserting casts or restricting to specific implementations, we can make interesting observations about the behavior of our direct semantics, as proven in the supplementary material. The first is that typed expressions are guaranteed to be either terminal or reducible:

Theorem 6.1 (Progress). *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_S \bar{\Psi}$ hold,*

$$\forall e, \tau. \quad \Psi \vdash e \; S \; \tau \quad \Longrightarrow \quad \Psi \vdash e \; \mathbf{terminal} \; \tau \quad \mathit{xor} \quad \exists e'. \quad \bar{\Psi} \vdash e \; R \; e'$$

where S is either optimistic $\triangleleft$ or pessimistic $\blacktriangleleft$ subtyping, and R is either optimistic $\twoheadrightarrow$ or pessimistic $\rightarrow$ reduction.

Note that this theorem states that even an *optimistically* typed expression is either terminal or *pessimistically* reducible. That is, we can even guarantee pessimistic progress for optimistic expressions. Also, note that in order to be **`terminal`**, every relevant value in v must be *pessimistically* typed. This is ensurable even for optimistically typed expressions because every optimistically typed *value* is necessarily also pessimistically typed.

The second observation we can make is that *pessimistic* typing is preserved by reduction:

Theorem 6.2 (Pessimistic-Type Preservation). *For every environment Ψ and implementation $\bar{\Psi}$ where $\vdash \Psi$ and $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ hold,*

$$\forall \tau, e, e'. \quad \Psi \vdash \tau \quad \wedge \quad \Psi \vdash e \blacktriangleleft \tau \quad \wedge \quad \bar{\Psi} \vdash e \; R \; e' \quad \Longrightarrow \quad \Psi \vdash e' \blacktriangleleft \tau$$

where R is either optimistic $\twoheadrightarrow$ or pessimistic $\rightarrow$ reduction.

Importantly, this states that even optimistic reduction preserves pessimistic typing, which is arguably the whole purpose of pessimistic typing. However, neither form of reduction preserves optimistic typing. Clearly optimistic reduction does not preserve optimistic typing, otherwise we would not be referring to it as *optimistic* typing. But it is surprising that even pessimistic reduction fails to preserve optimistic typing despite the many run-time assertions it makes. To see why, optimistically type the program **`let dynamic`** x := `"Hello"` **`in`** x `% 10`, and then try to optimistically type the reduction of that program, `"Hello" % 10`. This failure of pessimistic reduction is critical, as it illustrates why inserting casts is necessary to ensure soundness.

The third and final observation we make is that optimistic and pessimistic reduction *coincide* for *pessimistically* typed programs:

Theorem 6.3 (Pessimistic Identification). *For every environment* Ψ *and implementation* $\bar{\Psi}$ *where* $\vdash \Psi$ *and* $\Psi \vdash_{\blacktriangleleft} \bar{\Psi}$ *hold,*

$$\forall e, \tau. \quad \Psi \vdash e \blacktriangleleft \tau \quad \Longrightarrow \quad \forall v. \quad \bar{\Psi} \vdash e \twoheadrightarrow^{\infty} v : \tau \quad \Longleftrightarrow \quad \bar{\Psi} \vdash e \rightarrow^{\infty} v : \tau$$

This means that, for pessimistically typed programs, we can use the more efficient optimistic reduction and yet still enjoy the stronger guarantees of pessimistic reduction. In particular, a pessimistically typed program will never become unacceptably stuck by either semantics, so its observable results are completely described by its set of valuations, which is identical across the two forms of reduction. Again, this is not true for optimistically typed programs. Thus, given an optimistically typed program, we would like a way to interpret it using a "better"-behaved pessimistically typed program. This is the purpose of cast insertion, or program refinement, which we discuss next.

7 THE CAST SEMANTICS

We define the cast semantics for our gradual calculus using *program refinement*. Program refinement is a generalization of cast insertion, the process traditionally used to enforce soundness for gradual type systems [Findler and Felleisen 2002; Siek and Taha 2006; Tobin-Hochstadt and Felleisen 2006]. Whereas cast insertion traditionally specifies how to transform a program by inserting casts, program refinement simply states that two programs are similar but with one having some casts inserted, akin to the similarity relation defined by Tobin-Hochstadt and Felleisen [2006]. That is, refinement specifies no strategy about how to insert casts. A refinement might have too few casts to achieve a particular goal, or more casts than are strictly necessary. This laxity actually makes it easier to reason about refinement, especially with respect to reduction, and in a more uniform manner, especially with respect to typing.

Program Refinement. Program refinement is formalized using the judgement $\Psi \vdash e \preceq \tilde{e} : \tau$, which indicates that the expression $\tilde{e}$[2] is a refinement of e when the expected output type is τ. The formalization of refinement has only one interesting rule, presented below; the other rules in the supplementary material simply allow this rule to be applied throughout the program.

$$\frac{\Psi \vdash e \preceq \tilde{e} : \tau}{\Psi \vdash e \preceq \textbf{cast}\ \tilde{e}\ \textbf{to}\ \tau : \tau}$$

This rule is the only rule that lets refinement insert a cast. It states that we can refine a program by inserting a cast to the expected return type τ of the program. By restricting inserted casts to be of precisely this form, we ensure that they only check optimistic assumptions of the original program. In particular, we avoid inserting casts that would introduce run-time errors that have no relationship to the optimism of the original program, say by arbitrarily inserting casts of string expressions to integers.

Program Translation. We mentioned that refinement is reflexive, but the primary purpose of refinement is translation of optimistically typed programs into pessimistically typed programs. Although refinement does not specify how precisely to implement such a translation, we can combine it with the concepts we have already developed to formalize the concept of a translation. Given an environment definition Ψ and implementation $\bar{\Psi}$, expressions e and $\tilde{e}$, and type τ, we say we have a well-formed translation if $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ holds, as defined in Figure 7. That is, a translation is well-formed if the original program $\Psi \mid e$ *optimistically* has type τ, the translated

[2]Note that, whereas the grammar for a $\bar{\Psi}$ is different than that for a Ψ, the notation $\tilde{e}$ is not introducing a new grammar. It is simply a convention we employ to help the reader keep track of which expressions are "original" expressions versus "refined" expressions.

$$\frac{\vdash \Psi \qquad \Psi \vdash \tau \qquad \Psi \vdash e \triangleleft \tau \qquad \Psi \vdash_{\blacktriangleleft} \bar{\Psi} \qquad \Psi \vdash e \preceq \tilde{e} : \tau \qquad \Psi \vdash \tilde{e} \blacktriangleleft \tau}{\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau} \qquad \frac{\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau \qquad \bar{\Psi} \vdash \tilde{e} \twoheadrightarrow^{\infty} v : \tau}{\Psi \vdash e \rightsquigarrow^{\infty} v : \tau}$$

Fig. 7. Cast Semantics

program $\bar{\Psi} \mid \tilde{e}$ is a *refinement* of the original program with expected return type τ, and the translated program *pessimistically* has type τ.

This indicates when we have *a* well-formed translation, but for a given Ψ and e there may be multiple such translations. To this end, we have the following property, proven in the supplementary material, that all well-formed translations are semantically equivalent (recalling that pessimistically typed programs cannot get stuck in an unacceptable manner):

Theorem 7.1 (Translation Irrelevance). *For every* Ψ, $\bar{\Psi}_1$, $\bar{\Psi}_2$, e, $\tilde{e}_1$, $\tilde{e}_2$, *and* τ,

$$\begin{pmatrix} \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_1 \mid \tilde{e}_1 : \tau \\ \vdash \Psi \mid e \rightsquigarrow \bar{\Psi}_2 \mid \tilde{e}_2 : \tau \end{pmatrix} \implies \forall v. \quad \bar{\Psi}_1 \vdash \tilde{e}_1 \; R^{\infty} \; v : \tau \iff \bar{\Psi}_2 \vdash \tilde{e}_2 \; R^{\infty} \; v : \tau$$

where R *is either optimistic* $\twoheadrightarrow$ *or pessimistic* $\rightarrow$ *reduction.*

This means that, in order to use well-formed translation as a basis for our cast semantics, we just need some well-formed translation for our given optimistically typed program. Which one we happen to choose is irrelevant. Fortunately, we have the following:

Theorem 7.2 (Translation Existence). *For every environment* Ψ, *expression* e, *and type* τ,

$$\vdash \Psi \quad \wedge \quad \Psi \vdash \tau \quad \wedge \quad \Psi \vdash e \triangleleft \tau \quad \implies \quad \exists \bar{\Psi}, \tilde{e}. \quad \vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$$

Thus every optimistically typed program has a well-formed translation. Defining such a translation is straightforward and tedious, so we defer formal construction to the supplementary material.

Given that we have both translation irrelevance and existence, we can define the cast semantics for our gradually typed language using the judgement $\Psi \vdash e \rightsquigarrow^{\infty} v : \tau$ defined in Figure 7.

Semantic Preservation. Now that we know that we can always refine an optimistically typed program into a pessimistically typed program, we want to know that this translation respects the *direct* semantics of the original program in a reasonable manner. We demonstrate this with two observations, the proofs of which can be found in the supplementary material.

Theorem 7.3 (Pessimistic-Valuation Preservation). *For every environment* Ψ, *expression* e, *and type* τ *where* $\vdash \Psi$ *and* $\Psi \vdash \tau$ *and* $\Psi \vdash e \triangleleft \tau$ *hold,*

$$\forall v. \quad \Psi \vdash e \rightarrow^{\infty} v : \tau \quad \implies \quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau$$

This states that if the direct semantics of our original program can pessimistically produce some result, then translation also produces that result. That is, translation preserves pessimistic valuations. Note that translation does not preserve optimistic valuations, though. This is because a program can happen to optimistically reduce to some value even if it requires repeatedly violating expected invariants of the system throughout the process, and a typical sound gradual type system has no principled way of safely arriving at that haphazard but fortuitous result.

This leads us to wonder what happens when the original program goes awry. In particular, due to pessimistic progress and preservation, we know that the translation must result in some valuation even if the original program does not. The following gives us some insight into what the valuation must be.

Theorem 7.4 (Optimistic-Valuation Reflection). *For every environment Ψ, expression e, and type τ where $\vdash \Psi$ and $\Psi \vdash \tau$ and $\Psi \vdash e \triangleleft \tau$ hold,*

$$\forall v.\quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau \quad\Longrightarrow\quad \begin{array}{c} \Psi \vdash e \mathrel{\rightarrow\!\!\triangleright}^{\infty} v : \tau \\ \textit{or} \\ \Psi \vdash v\ \mathbf{bad\text{-}cast} \quad\wedge\quad \Psi \vdash e \mathrel{\rightarrow\!\!\triangleright}^{*} \mathbf{lapse}\ \tau \end{array}$$

This states that any valuation resulting from translation must also result optimistically from the original program *unless* the valuation is a bad cast catching the fact that the original program would become pessimistically stuck in an unacceptable manner, which Theorem 6.1 guarantees can only happen if the original program would become ill-typed. In combination with pessimistic-valuation preservation, this informs us that the cast semantics is essentially the same as the direct semantics *except* that it results in bad casts rather than lapsing.

Thus, with the combination of valuation preservation and reflection, we see that there is a very close relationship between our cast semantics and our direct semantics, one that is common among sound gradual type systems. This suggests that programmers can rely on the more intuitive direct semantics as a reasonable approximation of what the cast semantics provides. There is still some gap, though, since the cast semantics preserves *pessimistic* valuations but reflects *optimistic* valuations. In most gradual type systems, bridging this gap requires understanding the details of where casts are inserted and how they are implemented. In our system, though, we can actually close that gap. The stronger guarantees in the next section ensure that our cast semantics even reflects pessimistic valuations, showing that programmers need only understand direct pessimistic reduction to anticipate the behavior of our cast semantics.

8 THE GUARANTEES

The challenge at hand is to design a gradually typed language that is both principled and efficient. Here we address the principles by formalizing the guarantees that our calculus provides, the proofs of which can be found in the supplementary material. Afterwards, we will address efficiency by comparing with other similarly principled gradual type systems.

8.1 Immediacy

Sound gradual typing guarantees that a cast will fail before the program would get stuck in an unacceptable manner. However, most sound gradually typed languages only have this property with respect to optimistic reduction. Our system has a stronger property, which we call *immediacy*, formalized as follows:

Theorem 8.1 (Immediacy). *For every Ψ, $\bar{\Psi}$, e, $\tilde{e}$, and τ where $\vdash \Psi \mid e \rightsquigarrow \bar{\Psi} \mid \tilde{e} : \tau$ holds,*

$$\forall e'.\quad \begin{pmatrix} \Psi \vdash e \mathrel{\rightarrow}^{*} e' \\ \Psi \vdash e'\ \mathbf{lapse}\ \tau \end{pmatrix} \quad\Longrightarrow\quad \exists \tilde{e}'.\quad \begin{pmatrix} \bar{\Psi} \vdash \tilde{e} \mathrel{\rightarrow\!\!\triangleright}^{*} \tilde{e}' \\ \bar{\Psi} \vdash \tilde{e}'\ \mathbf{bad\text{-}cast} \end{pmatrix} \quad\wedge\quad \Psi \vdash e' \preceq \tilde{e}' : \tau$$

In the statement of this theorem, we distinguish the clause $\Psi \vdash e' \preceq \tilde{e}' : \tau$. Without this clause, the theorem simply states that the cast semantics results in a bad cast *whenever* the original program would eventually get pessimistically stuck in an unacceptable manner. This is sufficient to strengthen optimistic-valuation reflection into pessimistic-valuation reflection, as we discussed in the previous section. And with the distinguished clause, the theorem furthermore guarantees that the bad cast occurs *immediately* when the original program would get pessimistically stuck.

This is in contrast with most work on sound gradual typing. To see why, consider the following traditional gradually typed program:

$$\mathbf{let}\ \mathbf{dyn} \to \mathbf{dyn}\ f := (\lambda s : \mathbf{str}.\ s.\mathsf{length})\ \mathbf{in}\ \mathbf{let}\ \mathbf{int} \to \mathbf{int}\ g := f\ \mathbf{in}\ \mathit{slow}();\ g\ 5$$

This program can pessimistically reduce in a single step to the following:

$$\textbf{let}\ \textbf{int} \to \textbf{int}\ g := (\lambda s : \textbf{str}.\ s.\mathsf{length})\ \textbf{in}\ \mathit{slow}();\ g\ 5$$

This reduced program, however, can no longer reduce pessimistically. The value $\lambda s : \textbf{str}.\ s.\mathsf{length}$ fails to have the expected type $\textbf{int} \to \textbf{int}$ of the variable g, even optimistically. This clearly indicates a violation of the intended invariants of the program. For a gradual type system to provide immediacy, the cast semantics for this program would have to raise an error at this point in the execution. However, most prior work cannot recognize the error until the call to $g\ 5$ eventually executes.

Interestingly, threesomes [Siek and Wadler 2010] do raise an error immediately for this example, provided one uses a variant that is what Siek et al. [2009] describe as the *eager* error-detection strategy. Furthermore, it has been proven that eager threesomes can be viewed as a cast-insertion implementation of the semantics prescribed by Garcia et al. [2016] when applied to a gradually typed lambda calculus [Toro and Tanter 2017]. So it might generally be the case that the semantics prescribed by Garcia et al. [2016] will always provide immediacy.

8.2 Immediate Accountability

Accountability is the ability to indicate what component of the program is to blame for a given cast failure observed by the cast semantics of a program, and to furthermore ensure that only dynamically typed components are ever blamed. Like in previous work on blame [Ahmed et al. 2011; Tobin-Hochstadt and Felleisen 2006; Wadler and Findler 2009], we can augment our calculus with labels and errors so that, when such a cast failure occurs, it provides a label specifying some optimistic assumption that turned out not to hold at run time. However, we forgo such an augmentation here because, for our calculus, the process is particularly uninteresting.

The reason is that our system is transparent—unlike in most existing accountable systems, casts are not introduced by our operational semantics. This means that casts are only introduced by program refinement and so directly correspond to locations in the original program. All erroneous casts in our semantics have the property that they are casts to a class or interface type, never to **dynamic**. Program refinement only introduces casts of an expression to its expected return type, which means the receiver of such a cast must be statically typed. Furthermore, the expression being refined optimistically has that expected return type. If that expression were also statically typed, that would imply the expression also has that expected return type pessimistically. Type preservation would then ensure that this cast would succeed. So the cast can only fail if the expression is dynamically typed. Thus, all erroneous casts not in the original program are necessarily casts from dynamically typed code to statically typed code that were directly inserted by program refinement, making blame trivial to achieve.

But whereas accountability is the property that a failing cast correctly identifies a faulty optimistic assumption in the source code, what we call *immediate* accountability furthermore demands that execution is currently at that point in the source code. That is, optimistic checks either fail immediately or never. This property makes blame tracking completely unnecessary, since immediate accountability guarantees that a cast fails only if that cast itself is to blame. For our system, the reasoning above, in combination with immediacy, ensures that our system provides immediate accountability.

However, in general the combination of immediacy and accountability is not sufficient to provide immediate accountability. This is evidenced by the fact that eager threesomes [Toro and Tanter 2017] require blame tracking in order to provide accountability [Siek and Wadler 2010] even though they provide immediacy.

$$\frac{}{\top \sqsubseteq \top} \qquad \frac{}{C \sqsubseteq C} \qquad \frac{}{\tau \sqsubseteq \mathbf{dynamic}} \qquad \frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Gamma \sqsubseteq \Gamma' \qquad \tau \sqsubseteq \tau'}{\Gamma, \tau\ x \sqsubseteq \Gamma', \tau'\ x}$$

$$\frac{}{\infty \sqsubseteq \infty} \qquad \frac{}{x \sqsubseteq x} \qquad \frac{\tau \sqsubseteq \tau' \qquad e_1 \sqsubseteq e_1' \qquad e_2 \sqsubseteq e_2'}{\mathbf{let}\ \tau\ x := e_1\ \mathbf{in}\ e_2 \sqsubseteq \mathbf{let}\ \tau'\ x := e_1'\ \mathbf{in}\ e_2'} \qquad \frac{\forall i.\ e_i \sqsubseteq e_i'}{C(e_1, \ldots, e_n) \sqsubseteq C(e_1', \ldots, e_n')}$$

$$\frac{e \sqsubseteq e' \qquad \delta \sqsubseteq \delta'}{e.f_\delta \sqsubseteq e'.f_{\delta'}} \qquad \frac{e \sqsubseteq e' \qquad \delta \sqsubseteq \delta' \qquad \forall i.\ e_i \sqsubseteq e_i'}{e.m_\delta(e_1, \ldots, e_n) \sqsubseteq e'.m_{\delta'}(e_1', \ldots, e_n')} \qquad \frac{e \sqsubseteq e' \qquad \tau \sqsubseteq \tau'}{\mathbf{cast}\ e\ \mathbf{to}\ \tau \sqsubseteq \mathbf{cast}\ e'\ \mathbf{to}\ \tau'}$$

$$\frac{}{\cdot \sqsubseteq \cdot} \qquad \frac{\Psi \sqsubseteq \Psi' \qquad i \sqsubseteq i'}{\Psi, i \sqsubseteq \Psi', i'} \qquad \frac{\Psi \sqsubseteq \Psi' \qquad c \sqsubseteq c'}{\Psi, c \sqsubseteq \Psi', c'}$$

$$\frac{\forall i.\ \tau_i \sqsubseteq \tau_i' \qquad \forall i.\ d_i \sqsubseteq d_i'}{\mathbf{class}\ C(\tau_1\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1; \ldots\} \sqsubseteq \mathbf{class}\ C(\tau_1'\ f_1, \ldots)\ \mathbf{implements}\ C_1, \ldots\ \{d_1'; \ldots\}}$$

$$\frac{\forall i.\ s_i \sqsubseteq s_i'}{\mathbf{interface}\ C\ \{s_1; \ldots\} \sqsubseteq \mathbf{interface}\ C\ \{s_1'; \ldots\}} \qquad \frac{\tau \sqsubseteq \tau' \qquad \Gamma \sqsubseteq \Gamma'}{\tau\ m(\Gamma) \sqsubseteq \tau'\ m(\Gamma')} \qquad \frac{s \sqsubseteq s' \qquad e \sqsubseteq e'}{s \mapsto e \sqsubseteq s' \mapsto e'}$$

Fig. 8. Optimism Relation, a.k.a. Precision Relation [Garcia et al. 2016; Siek et al. 2015a]

8.3 The Gradual Guarantee

The gradual guarantee [Siek et al. 2015a], in our terms, states that adding optimism to a program should increase the likelihood that the program will type-check and evaluate successfully, and nothing more. We formalize this using an *optimism* relation ($\sqsubseteq$), shown in Figure 8, which indicates when two components only differ in terms of degree of optimism, with the right component being the more optimistic of the two. This is traditionally known as a precision relation [Garcia et al. 2016; Siek et al. 2015a] or naïve subtyping [Wadler and Findler 2009]. We use the new terminology both to be consistent to with the rest of the paper and to address the fact that the precision relation is backwards, as noted by its inventors [Siek et al. 2015a], since it places the more precise component on what the name suggests should be the less precise side.

The gradual guarantee formally consists of three theorems adapted from [Siek et al. 2015a]. Our first theorem states that a program that is already optimistically typed will still be optimistically typed if it is made more optimistic:

Theorem 8.2 (Gradual Optimism).

$$\forall \begin{pmatrix} \Psi, \Psi' \\ \Gamma, \Gamma' \\ \tau, \tau' \\ e, e' \end{pmatrix}. \quad \begin{pmatrix} & \vdash \Psi \\ \Psi & \vdash \Gamma \\ \Psi & \vdash \tau \\ \Psi \mid \Gamma & \vdash e \triangleleft \tau \end{pmatrix} \wedge \begin{pmatrix} \Psi \sqsubseteq \Psi' \\ \Gamma \sqsubseteq \Gamma' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \end{pmatrix} \implies \begin{pmatrix} & \vdash \Psi' \\ \Psi' & \vdash \Gamma' \\ \Psi' & \vdash \tau' \\ \Psi' \mid \Gamma' & \vdash e' \triangleleft \tau' \end{pmatrix}$$

Our second theorem states that if a program results in a valuation, then a more optimistic version of that program results in the same valuation or some more optimistic one *unless* the valuation was an overly pessimistic cast in the more pessimistic program.

Theorem 8.3 (Gradual Preservation). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall v. \quad \begin{pmatrix} \Psi \vdash e \rightsquigarrow^{\infty} v : \tau \\ \Psi \sqsubseteq \Psi' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \end{pmatrix} \implies \exists v'. \quad \begin{pmatrix} \Psi' \vdash e' \rightsquigarrow^{\infty} v' : \tau' \\ v \sqsubseteq v' \\ \textit{or} \\ \Psi \vdash v \ \textbf{bad-cast} \end{pmatrix}$$

Our third theorem states that, if an optimistic program results in a valuation, then a more pessimistic version results in that same valuation or some more pessimistic one *unless* it encounters an overly pessimistic cast first.

Theorem 8.4 (Gradual Reflection). *For every* Ψ, Ψ', e, e', τ, *and* τ' *such that* $\vdash \Psi$, $\vdash \Psi'$, $\Psi \vdash \tau$, $\Psi' \vdash \tau'$, $\Psi \vdash e \triangleleft \tau$, *and* $\Psi' \vdash e' \triangleleft \tau'$ *hold,*

$$\forall v'. \quad \begin{pmatrix} \Psi \sqsubseteq \Psi' \\ \tau \sqsubseteq \tau' \\ e \sqsubseteq e' \\ \Psi' \vdash e' \rightsquigarrow^{\infty} v' : \tau' \end{pmatrix} \implies \exists v. \quad \Psi \vdash e \rightsquigarrow^{\infty} v : \tau \quad \wedge \quad \begin{pmatrix} v \sqsubseteq v' \\ \textit{or} \\ \Psi \vdash v \ \textbf{bad-cast} \end{pmatrix}$$

Together, these theorems prove our calculus provides the gradual guarantee. Interestingly, gradual preservation and reflection can be derived from our earlier theorems by making one key observation: making a program more optimistic has the effect of making it more likely to be able to reduce pessimistically. Thus our direct semantics provides new perspective on the gradual guarantee.

8.4 Transparency

Lastly, it is easy to prove the following theorem about our optimism relation:

Theorem 8.5 (Transparency).

$$\forall v, v'. \quad v \sqsubseteq v' \quad \implies \quad v = v'$$

For languages providing the gradual guarantee, we believe this accurately formalizes our concept of transparency. In particular, the combination implies that making a program more optimistic will not affect the values that arise during that program's execution. This is in contrast to calculi like the cast calculus [Siek et al. 2015a], in which two values can be related and yet differ due to inserted casts, which are precisely the wrapper functions we actively avoided in order to get the following promising experimental results.

9 EXPERIMENTAL EVALUATION

We claim that our approach to gradual typing can be implemented efficiently and avoid the performance pitfalls of gradual typing that Takikawa et al. [2016] described. Here we present an evaluation of our experimental language called Nom. We used benchmarks from two different sources: first, there are two benchmarks from the benchmark suite used by Takikawa et al. [2016], and second, there are five benchmarks that are among those that Vitousek et al. [2017] selected from the official Python benchmark suite [Python Development Team 2008] at the time. These serve to evaluate our implementation on two metrics, respectively. The first set of benchmarks tests the overhead that is introduced at the boundaries between typed and untyped code. The second set of benchmarks tests whether type annotations improve the performance of programs, which is a part of our motivation for gradual typing. For comparison with another sound nominally typed language with gradual typing, we also translated the first group of benchmarks to C#, and we present the results of running those translations alongside the others.

9.1 The Experimental Compiler

Our experimental compiler supports our language Nom that implements the formalized features discussed so far along with mutable state, primitive types, implementation inheritance, overloading, access/visibility modifiers, and static fields and methods. Unlike in our calculus, field accesses and method invocations are not explicitly annotated with a dispatch mode, and the supplementary material discusses how Nom addresses the subtleties involved in bridging this gap.

Because dynamic checks are more common with gradual typing, we make some optimizations to the standard implementation for a nominally typed object-oriented language. At compile time, a number is generated for each class type. An object is represented as its class number followed by its fields. The class number is used to index arrays that provide standard features such as method tables and interface tables, which are used by statically typed method invocations. Each class index is also associated with a flat list of all its supertypes—class hierarchies are usually rather shallow, so scanning these lists for a matching supertype can be expected to be quick. In order to make dynamically typed method invocations efficient, the class number is used to access an array of association lists mapping method identifiers to dispatching methods, each of which employs a statically determined decision tree to determine which overloading to call, if any, based on the types of the arguments. This is essentially an extension of the hybrid-casting technique of Allende et al. [2013] in GradualTalk [Allende et al. 2014]. Furthermore, at the call site of each applicable method invocation, we cache the result of method lookup for the three most recent run-time types of the receiver. This is a standard technique for dynamic languages, known as inline caching [Ahn et al. 2014; Deutsch and Schiffman 1984].

Rather than compiling to assembly, our compiler translates to C, which can then be compiled by a standard C compiler.[3] We use the Boehm-Demers-Weiser conservative garbage collector [Demers et al. 1990].

9.2 Design of Benchmark Programs

In contrast to work that adds gradual typing to existing programming languages, we do not have access to a large collection of programs written in our language. However, as a first step, all we need is a program that has a large number of transitions between untyped and typed code, as these are the only possible sources of gradual-typing overhead in our system. Fortunately, the two smallest poorly performing (i.e. more than 100x slowdown) programs in the benchmark suite of Takikawa et al. were also among those with the highest numbers of boundary transitions. These two programs are `sieve` and `snake`. `sieve` implements the sieve of Eratosthenes using streams to determine the $10{,}000^{\text{th}}$ prime number. `snake` implements the popular game Snake and runs it using a statically predetermined list of about 55,000 moves and events. Note that `sieve` in particular was written "to illustrate the pitfalls of sound gradual typing" [Takikawa et al. 2016], as it consists of just two heavily interacting modules.

Given that the programs were written in a different programming paradigm, there are some design choices to be made in how to translate them to Nom and C#. We strove to mimic the structure of the original programs as much as possible in order to keep the numbers and kinds of transitions across module boundaries the same. The biggest differences are that we manually implement tail-recursion elimination in our translation and—as Nom does not support anonymous functions—we model function types using interfaces and closures using classes. All in all, the converted programs are nominal but not necessarily written in an object-oriented style. Thus good performance with these programs is likely to imply good performance in most cases both because

[3] For the benchmarks, we use the Microsoft C compiler, set to optimize for speed (`/O2`).

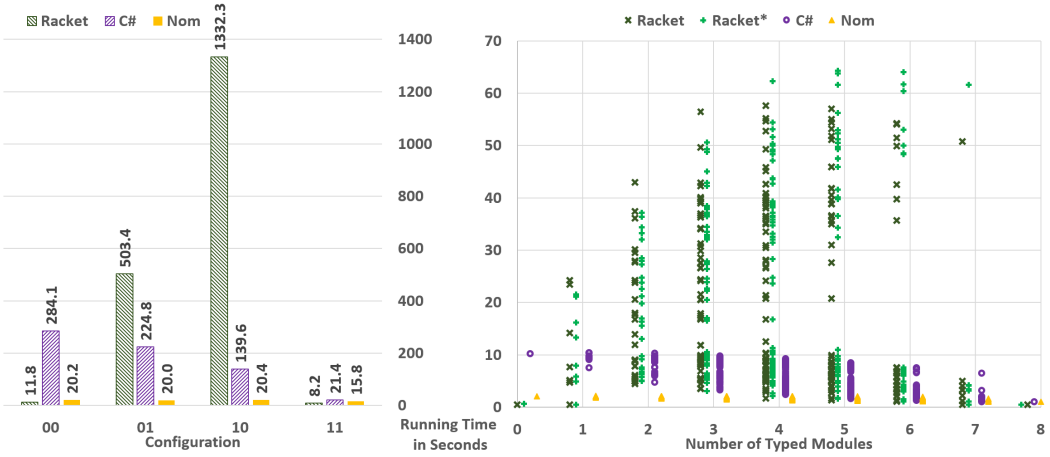

Fig. 9. Benchmark results for `sieve` (left) and `snake` (right)

they have already been demonstrated to cause problems for prior work due to frequent interaction between modules and because they are written in a style that is not favored by our implementation.

The Python benchmarks were much easier to translate, as they were written in a style that fits our language much more closely. As such, they are more what a typical benchmark for our language would look like.

9.3 Benchmark Results

All benchmarks were run on an Intel Core i7-3770 at 3.4Ghz with 16GB of main memory, running Windows 7 with minimal background activity. The benchmark programs were run over several iterations. For each iteration, the sequence in which individual configurations were run was determined randomly.

9.3.1 Sieve. `sieve` is an extreme microbenchmark, consisting of just two heavily interacting modules with several hundred million transitions between those two modules. As such, it is a key benchmark to measure the efficiency of casts in a gradually typed language. The left-hand side of Figure 9 shows the results for the `sieve` benchmark for Racket, C#, and Nom. There are four configurations, corresponding to the fully untyped program "00", the fully typed program "11", and the two mixed configurations "01" and "10". In Typed Racket, the two mixed configurations cause extreme overheads due to gradual typing, as described by Takikawa et al. [2016]. C#, on the other hand, is unaffected by interaction but instead suffers significant slowdown in the presence of dynamic typing.

Regarding Nom, its performance is, in relative terms, fairly constant across the configurations, though there is an increase in performance in the fully typed configuration. This is despite the fact that we still measured several hundreds of millions of transitions between typed and untyped code when executing either mixed configuration in Nom, the same magnitude that Takikawa et al. reported for Racket.

9.3.2 Snake. The right-hand side of Figure 9 shows the timings for `snake` as a scatter plot of running times, in seconds, grouped by the number of typed modules. There are two versions for Racket here because the original version published by Takikawa et al. checks the entire contents of lists when casting them from untyped code to typed code, an operation that in theory increases the

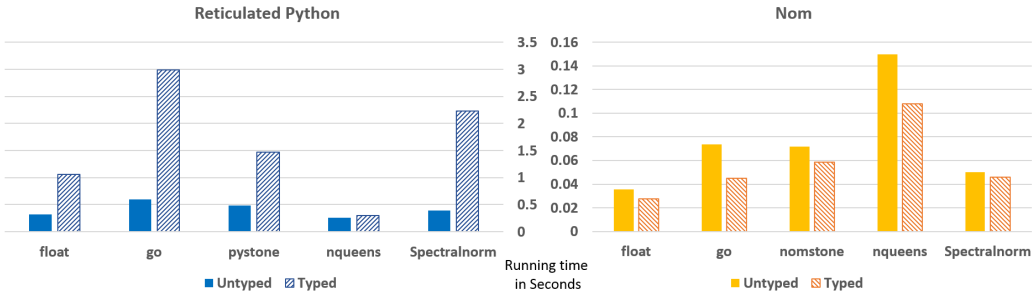

Fig. 10. Benchmarks taken from Vitousek et al. [2017]'s selection of Python benchmarks

time complexity of the programs. We thus developed a modified version of `snake`, labeled Racket*, that uses a user-defined structure instead of Racket's `cons`-lists in an attempt to make those checks lazy, similar to how our Nom implementation of lists works. Interestingly, there does not seem to be much difference in performance between the two Racket versions, suggesting that the performance issues Takikawa et al. observed are due to the concerns we have discussed throughout the paper rather than due to the use of deep casts. As before, the performance of Nom, on the other hand, consistently improves as more types are added to the program. The same holds for C#, though again suffering significantly more overhead in the presence untyped code.

9.3.3 Python Benchmarks. For the Python benchmarks, we chose five with some preference towards the ones that had poor performance under the *transient*-cast implementation of Vitousek et al. [2017] (`pystone` and `float` suffer from about 200% overhead, and `go` and `spectralnorm` suffer from about 400% overhead for *typed* code compared to untyped code[4]). In contrast to the Racket benchmarks, these programs were written in a language that is close to ours and thus were translated with minimal effort. The left-hand side of Figure 10 shows the results of these benchmarks for Reticulated Python, and the right-hand side of Figure 10 shows the results for Nom. The absolute running times should not be compared other than to serve as an indicator of overall reasonableness; Python is interpreted, whereas our code is compiled and optimized by a C compiler, so absolute differences are not meaningful. The effect of types on performance within each language is meaningfully different, though. The transient casting strategy slows down programs as more type annotations are added because type annotations cause checks to be inserted and executed regardless of whether the whole program is typed or not. This may be a reasonable thing to do in the scenarios that Vitousek et al. are considering, where an open world can readily circumvent invariants of the gradual-typing implementation, but we believe that in general programs should overall become faster as more type annotations are added due to the additional optimizations this enables. Nom achieves this goal here, although the `snake` benchmark best illustrates this behavior because it provides data on many intermediate configurations as well.

9.4 Validity

We only evaluated our system on a small set of small programs. While our system performed well for these programs, there is always the possibility that it may perform poorly for some other program. However, by the nature of our implementation, our overhead is proportional to the number of run-time interactions between typed and untyped code. Importantly, our overhead is fairly unaffected by the kind of interactions that occur due to the transparency of our casts. Consequently, it is likely

[4]Without blame. Adding blame tracking in many cases more than doubles the overhead.

the case that `sieve` does in fact represent a worst-case scenario regarding overhead created by our system due to the immense degree of interaction points as designed by Takikawa et al. While it seems possible that there are other programs that could increase our overhead by small factors, say by designing a program to thwart any effectiveness of inline caches, it seems unlikely that there are programs that would increase our overhead by large factors, especially to the degree observed in the related works we compare to.

As for the measurements we do provide, the usual caveats to experimental running-time measurements apply. Efforts we took to mitigate the risk of obtaining misleading numbers include

- running the benchmarks sequentially, not in parallel, with a separate randomized order for each trial run,
- confirming that several minor variations of the Nom benchmarks, such as employing object-oriented-style dynamic dispatch instead of functional-style static dispatch, exhibited similar performances,
- and observing no significant differences in performance across three different machines.

Furthermore, the artifact-evaluation committee approved the validity of our manual translations of the benchmarks and successfully replicated our results.

10 DISCUSSION

To the best of our knowledge, we have provided the first results of an evaluation of a sound gradually typed language following the methodology suggested by Takikawa et al. [2016] that shows minimal overhead. While our type system is relatively restrictive compared to other gradually typed languages, we hope to have established a baseline from which more expressive sound gradual type systems can be explored with similarly minimal overhead. In this section, we argue that this baseline is valid and useful, sketching an outlook of how to get to more expressive type systems from it, and sketching the challenges that still need to be overcome.

10.1 Designing for Performance

Nom's efficiency comes from a combination of multiple factors that keep potentially expensive run-time operations cheap. Nominality makes casts infrequent and efficient. Transparency prevents overhead due to wrapper allocation. Immediate accountability makes blame tracking unnecessary. Furthermore, the clear separation of dynamic vs. statically checked field accesses and method invocations allows us to implement and optimize both using the techniques appropriate to each. In the following, we illustrate the advantages of each of these properties by comparing Nom to the other languages that we benchmarked.

Typed Racket. In Typed Racket, most of the overhead of gradual typing is caused by expensive run-time checks. Compared to Nom, these have two major causes: wrappers vs. transparency, and structural vs. nominal. In gradually-typed Racket, transferring a value across the typed/untyped boundary, in either direction, often requires the value to be wrapped in order to enforce soundness and provide blame. Thus each such transfer causes an allocation, and these wrappers themselves often have to produce more wrapped values, leading to more allocations. Allocation is known to be a fairly slow operation even in Racket where allocation is quite optimized. Furthermore, these wrappers introduce layers of indirection, especially since wrappers often end up being stacked onto each other in our benchmarks. Note that this stacking is indirect, so using *threesomes* à la Siek and Wadler [2010] would not help—in fact it would only add more overhead. Our system is transparent, so we suffer none of these allocations or layers of indirection.

The second major reason is that Typed Racket uses a structural type system whereas we use a nominal type system. In Typed Racket, one can dynamically check that a value has a field of

the appropriate name, and one can check that the value in that field is a function. However, one cannot check what kind of function it is. Consequently, every time one uses that function to get an integer (the type that your typed code is expecting), one has to check that it actually returns an integer. This is even the case when that function was created by typed code but happened to transfer through untyped code. In Nom, we can often accomplish all these checks with a single nominal check. In particular, if the value was created by typed code, then that single nominal check accomplishes what Racket would need a potentially infinite number of structural checks for. In problematic programs, such as `sieve` and `snake`, these two major differences can each introduce multiple factors of overhead in gradually-typed Racket (but not in Nom), explaining the performance differences.

C#. We experimented with a few variations of dynamic programs in C# to investigate why increased dynamism causes large overheads. As an example, we experimented with casting everything to and from `Object` instead of `dynamic`. Conceptually, these two programs should have the same performance since the `dynamic` program is simply doing those casts implicitly at run time. However, we found that the `Object` version of the program was significantly faster. This leads us to believe that C#'s choice of implementation of `dynamic`, which recompiles the relevant code at run time using the run-time type of the relevant value, is the cause of its inefficiency. This choice seems to be forced upon C# in order to accommodate the many type-system features of C# that were not designed with gradual typing in mind, an issue we will discuss this further in Section 10.3.

Reticulated Python. As already stated in Section 9.3.3, the transient casting strategy in Reticulated Python inserts more casts as more type annotations are added, making fully typed code the slowest configuration of any program. In contrast, pessimistically typed code in Nom is sound without any casts being inserted and can additionally benefit from type-directed compiler optimizations. Furthermore, the numbers we give for the Reticulated Python benchmarks are for the versions of the programs where blame tracking was turned off. Blame tracking significantly increases—sometimes doubles—the overhead of gradual typing in Reticulated Python. In contrast, as discussed in Section 8.2, Nom's immediate accountability makes blame tracking completely unnecessary.

10.2 Scaling to Industry

Our compiler and language are experimental and thus lack many features that real-world compilers and languages would have, such as support for debugging, multithreading, separate compilation, etc. However, features that do not affect the type system should not affect the efficiency of gradual-typing-related operations. In fact, in contrast to systems with monotonic run-time type information, our approach has trivial multithreading support, as all operations during a cast are read-only. With respect to type-system features, the major differences between Nom and pre-generics Java is that Nom restricts overloading and does not support exceptions and arrays (though we provide a natively implemented `ArrayList` class whose getter method is typed as returning `dynamic`). We do not believe that adding these features would cause significant gradual-typing overhead. If that turns out to be the case, then Nom should be easy to extend to something that could be used in an industrial setting, although incorporating more powerful features such as generics would still require substantial research, as discussed next.

10.3 Increasing Expressiveness

Nominal typing faces many challenges specific to nominality. This is true even without gradual typing, and still today discoveries about the foundations of nominal typing are being made. Here we discuss some of the challenges related to gradual typing, in particular ones that make the gradual guarantee difficult to achieve.

10.3.1 Types Affect Execution. Unlike structurally typed languages, types in statically typed nominal languages often affect execution. Examples of this are method overloading and extension methods. With method overloading, the type of the arguments is used to determine which overloading to call. When there is ambiguity, languages like C# and Java report a type error forcing the programmer to resolve the ambiguity before compiling. This use of ambiguity as error is often necessary when types affect execution in order to keep execution predictable.

However, with gradual typing these ambiguities arise at run time, when the programmer is not generally available to disambiguate the execution, and so an error is thrown. And because the run-time types of values can provide more overloadings than might have been statically available, making a program more dynamic can even introduce such ambiguity errors, violating the gradual guarantee. Thus, gradually typed nominal languages cannot rely on the programmer to resolve the many ambiguities that statically typed nominal language often have. For method overloading, this means all overloadings provided by a given class or interface must have pairwise disjoint signatures in order to satisfy the gradual guarantee.

C# has another issue with gradual typing: extension methods. Extension methods are a way to retroactively add methods to an interface or class. In C#, when the type-checker fails to find a method declaration in a given receiver's class or interface, it checks for extension methods defined for that class or interface in the current static scope. But this faces two challenges when gradually typed. First, the current static scope is not available at run time. As a consequence, C# fails to identify extension methods at run time and simply throws a run-time type exception. This means that LINQ, the specialized syntax used to describe database queries that is built entirely on extension methods, is completely unusable by dynamically typed C#. Second, a method declaration that is not visible at compile time, so that the extension method is invoked, could be visible at run time, so that the instance's method is invoked, thereby causing dynamic typing to change the semantics of the program, violating the gradual guarantee.

10.3.2 Generics. Java, C#, and Scala have all had generics for over a decade, and more recent nominal languages such as Ceylon and Kotlin continue the trend. Thus gradual typing for nominal types needs to address generics. Ina and Igarashi have considered gradual typing for generics [Ina and Igarashi 2011]. They have an interesting approach to `Foo<dynamic>`, where `Foo<T>` is a generic class, which is to consider all uses of `T` in the body of `Foo` as potentially `dynamic`. Unfortunately, this means that even well-typed code may need to have frequent run-time checks inserted. Furthermore, they do not consider generic methods, type-argument inference, or any form of variance, all of which are essential to how generics are used in practice. As such, there is still significant work to be done for generics, some of the many challenges of which we discuss here.

One surprisingly simple yet challenging problem is decidability. In order to fulfill the gradual guarantee, subtyping needs to be decidable so that run-time casts are guaranteed to terminate. In the presence of variant generics, subtyping is often undecidable, as Kennedy and Pierce [2007] and Grigore [2017] have shown. A promising approach is that of Greenman et al. [2014], who identified a pattern in how generics are used in practice; a pattern that can be enforced by the language design to ensure that subtyping is decidable.

Our approach requires the ability to check subtyping at run time. This implies that every instance of `List<String>` stores the information necessary to determine at run time that the instance is not just a `List`, but a `List` of `String`. This is known as reified generics and is the counterpoint to type erasure. This might cause some concern, as reified generics imply that type information has to be constructed and passed around at run time throughout generic methods. Schinz [2005] did an analysis of what the impact of this would be for Scala on the JVM, and he found that it would on average make programs run 50% slower, allocate 140% more memory, and compile to 30% more

byte code. However, Microsoft added reification to the CLR because of its potential to improve performance with primitive types and specialization, and Ceylon's generics are reified because the team found they could implement it with little overhead, even on the JVM. It is thus unclear what the overhead of reified generics might actually entail for gradual typing.

The greatest challenge, though, is likely to be type-argument inference, a feature that is critical to making use of generic methods convenient. To understand why it is likely to be such a significant challenge, consider the following statically typed C# function `SnocS` (without extension methods):

```
List<T> SnocS<T>(IEnumerable<T> startS, T endS) {
  var elemsS = Enumerable.ToList(startS);
  elemsS.Add(endS);
  return elemsS;
}
```

and its corresponding dynamically typed C# function `SnocD`

```
dynamic SnocD(dynamic startD, dynamic endD) {
  var elemsD = Enumerable.ToList(startD);
  elemsD.Add(endD);
  return elemsD;
}
```

In order to fulfill the gradual guarantee, if a call to `SnocS` succeeds, the same call to `SnocD` should succeed. However, the expression `SnocS(new List<string>(), 5)` succeeds in C#, whereas the expression `SnocD(new List<string>(), 5)` throws a run-time type exception. In particular, the invocation `elemsD.Add(endD)` fails because at run time `elemsD` is a `List<string>` but `endD` is an `int`. In the corresponding line of `SnocS`, the run-time type of `elemsS` is `List<object>`. The cause of the difference in behavior is that in `SnocS`, the type argument for `ToList` is inferred to be `T`, which at run time is `object`, due to the static type of `startS` *and* `endS`, whereas in `SnocD` it is inferred to be `string` due to the dynamic type of `startD`. Thus, in addition to developing a decision procedure for type-argument inference, which is as of yet an unsolved problem, a gradual type system for generics must also overcome this challenge regarding the gradual guarantee.

It is due to these many complications with nominal typing that C# is forced to implement gradual typing using run-time compilation. This unfortunate fact is likely the cause of its poor performance in Section 9. Thus, with nominal typing, it seems to be important to design the language with gradual typing in mind in order to not only achieve the gradual guarantee, but also to achieve efficient implementation of dynamic typing.

10.3.3 Interacting with Structural Values. Our work relies heavily on one major assumption: that all *C*-like values can be explicitly tagged to indicate that they indeed are instances of *C*. It is particularly critical that this assumption applies even to values created in untyped code.

However, there are many situations in which structural values are either unavoidable or are the appropriate solution. Values might originate from other languages, with JavaScript being a particularly notable example. Records might be the natural way to represent certain data, and are especially useful for interacting with databases or performing database-like operations. Programmers might want to write function expressions without having to concern themselves with determining specifically which interface that function is implementing. Each of these are important applications of gradual typing that are especially well suited for structural typing and therefore especially challenging for nominal typing.

Structural Run-Time Type. One likely step towards addressing these challenges is to allow a `Structural` run-time type. In C#, this type is the specially handled `ExpandoObject` class. Values

of this type would be structural values, and method invocation and field access on this type would implement standard structural method invocation and field access. This would likely be a sufficient solution for records and interacting with databases. However, like Thorn's `like` types [Wrigstad et al. 2010] or StrongScript's `optional` types [Richards et al. 2015], it provides no means for structural values to be given to code expecting nominal values, implying there is still a significant interoperability barrier between the two.

One potential solution to overcoming this barrier is to adapt the existing work on monotonic references [Siek et al. 2015b; Swamy et al. 2014; Vitousek et al. 2014] to assign nominal type-tags to objects. A question that needs to be resolved here is whether additional type specialization should be allowed to happen after the first tag has been added, as this is not something one would expect from a nominal type system. Furthermore, although this avoids introducing wrappers, it is not fully transparent since it modifies values in place, so it is possible performance may return as an issue.

First-Class Functions. Given that it is in general impossible to decide whether a given function only returns values of some type when passed values of some other type, implementing transparent casting and immediate accountability for unannotated functions is a daunting task. Intraprocedural type inference is still an unsolved problem in object-oriented languages like ours, but if it is possible, it should enable run-time type-checking of dynamic code. In such a scenario, function values could then mostly consist of an AST representation and a map from already checked and validated signatures of the function to accordingly compiled code. In the absence of intraprocedural type inference, another solution might be to have declared function types, similar to C#'s delegates, and require that such a type be specified for every anonymous function. Alternatively, one might be able to efficiently adapt monotonic casting specifically to function values and nominal interfaces.

11 CONCLUSION

We have provided new properties of gradual type systems that, in conjunction with the gradual guarantee, capture an intuition about when and where gradual typing can produce overhead even in the ideal case. The properties do not necessarily guarantee an efficient implementation of gradual typing, as we demonstrate with benchmarks for C#.

We showed, however, that by codesigning the type system and underlying runtime system alongside these desired properties for gradual typing, we could create an efficient and well-behaved gradually typed nominal object-oriented language. We provided evidence that our language does not suffer from previously measured extreme overheads due to gradual typing, even in adversarial scenarios where programs have a high level of interaction between typed and untyped code.

As part of our design, we chose to use nominal typing instead of structural typing as an explicit tradeoff of expressiveness for performance. We argued how this loss of expressiveness is acceptable for many applications of gradual typing, and we illustrated paths forward towards recovering expressiveness while still maintaining performance. In general, there are many desirable features that our language does not have, but it seems that many of them can be added with reasonable effort. Indeed, sound gradual typing is *nominally* alive and well.

ACKNOWLEDGMENTS

This work was supported by the NSF under grant CCF-1350182. The authors would like to express their thanks to the anonymous reviewers, the artifact-evaluation committee, the members of IFIP WG 2.16, the Ceylon team, the Kotlin team, Benjamin Chung, Ronald Garcia, Ben Greenman, Andrew Hirsch, Tom Magrino, Matthew Milano, Greg Morrisett, Gregor Richards, Adrian Sampson, Jeremy Siek, Éric Tanter, and Sam Tobin-Hochstadt for their numerous thoughts and suggestions.

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *POPL 2011*. ACM, New York, NY, USA, 201–214.

Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *PLDI 2014*. ACM, New York, NY, USA, 496–507.

Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. 2014. Gradual Typing for Smalltalk. *Science of Computer Programming* 96 (2014), 52 – 69. Special issue on Advances in Smalltalk based Systems.

Esteban Allende, Johan Fabry, and Éric Tanter. 2013. Cast Insertion Strategies for Gradually-Typed Objects. In *DLS 2013*. ACM, New York, NY, USA, 27–36.

Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems are Unsound: The Existential Crisis of Null Pointers. In *OOPSLA 2016*. ACM, New York, NY, USA, 838–848.

Christopher Anderson and Sophia Drossopoulou. 2003. BabyJ: From Object Based to Class Based Programming via Types. In *WOOD 2003*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, 53 – 81.

Gavin Bierman, Erik Meijer, and Mads Torgersen. 2010. Adding Dynamic Types to C#. In *ECOOP 2010*. Springer Berlin Heidelberg, Berlin, Heidelberg, 76–100.

Gilad Bracha. 2004. Pluggable Type Systems. (2004). In *Workshop on Revival of Dynamic Languages*.

Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems. In *POPL 2016*. ACM, New York, NY, USA, 443–455.

Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. 1990. Combining Generational and Conservative Garbage Collection: Framework and Implementations. In *POPL 1990*. ACM, New York, NY, USA, 261–269.

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *POPL 1984*. ACM, New York, NY, USA, 297–302.

Facebook, Inc. 2016. The Hack Language Specification, Version 1.1. (April 2016).

Robert Bruce Findler and Matthias Felleisen. 2002. Contracts for Higher-Order Functions. In *ICFP 2002*. ACM, New York, NY, USA, 48–59.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *POPL 2016*. ACM, New York, NY, USA, 429–442.

Ben Greenman, Fabian Muehlboeck, and Ross Tate. 2014. Getting F-Bounded Polymorphism into Shape. In *PLDI 2014*. ACM, New York, NY, USA, 89–99.

Radu Grigore. 2017. Java Generics are Turing Complete. In *POPL 2017*. ACM, New York, NY, USA, 73–85.

Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. 2006. SAGE: Hybrid Checking for Flexible Specifications. *Scheme and Functional Programming Workshop* 6 (2006), 93–104.

Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22, 3 (1994), 197 – 230.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. 2001. Featherweight Java: A Minimal Core Calculus for Java and GJ. *TOPLAS* 23, 3 (May 2001), 396–450.

Lintaro Ina and Atsushi Igarashi. 2011. Gradual Typing for Generics. In *OOPSLA 2011*. ACM, New York, NY, USA, 609–624.

Andrew Kennedy and Benjamin C. Pierce. 2007. On Decidability of Nominal Subtyping with Variance. In *FOOL/WOOD 2007*. Microsoft Research, Cambridge, UK, 1–12.

Jacob Matthews and Robert Bruce Findler. 2007. Operational Semantics for Multi-Language Programs. In *POPL 2007*. ACM, New York, NY, USA, 3–10.

Microsoft. 2012. TypeScript. (Oct. 2012).

The Python Development Team. 2008. Python Benchmarks. https://hg.python.org/benchmarks/. (Dec. 2008).

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe & Efficient Gradual Typing for TypeScript. In *POPL 2015*. ACM, New York, NY, USA, 167–180.

Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. 2015. Concrete Types for TypeScript. In *ECOOP 2015*, Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 76–100.

Michel Schinz. 2005. *Compiling Scala for the Java Virtual Machine*. Ph.D. Dissertation. EPFL.

Jeremy Siek, Ronald Garcia, and Walid Taha. 2009. Exploring the Design Space of Higher-Order Casts. In *ESOP 2009*. Springer-Verlag, Berlin, Heidelberg, 17–31.

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007*. Springer-Verlag, Berlin, Heidelberg, 2–27.

Jeremy G Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. *Scheme and Functional Programming Workshop* 6 (2006), 81–92.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *SNAPL 2015*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 274–293.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *ESOP 2015*. Springer Berlin Heidelberg, Berlin, Heidelberg, 432–456.

Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without Blame. In *POPL 2010*. ACM, New York, NY, USA, 365–376.

Daniel Smith and Robert Cartwright. 2008. Java Type Inference is Broken: Can We Fix It?. In *OOPSLA 2008*. ACM, New York, NY, USA, 505–524.

Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *POPL 2014*. ACM, New York, NY, USA, 425–437.

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

Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage Migration: From Scripts to Programs. In *OOPSLA 2006*. ACM, New York, NY, USA, 964–974.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *SAS 2017*. Springer International Publishing, Cham, 382–404.

Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *DLS 2014*. ACM, New York, NY, USA, 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 *POPL 2017*. ACM, New York, NY, USA, 762–774.

Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can't Be Blamed. In *ESOP 2009*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. 2010. Integrating Typed and Untyped Code in a Scripting Language. In *POPL 2010*. ACM, New York, NY, USA, 377–388.