KafKa: Gradual Typing for Objects

Benjamin Chung, Paley Li, Francesco Zappa Nardelli & Jan Vitek

Northeastern University & INRIA & Czech Technical University

Abstract

The enduring popularity of dynamically typed languages has motivated research on *gradual type systems* to allow developers to annotate legacy dynamic code piecemeal. Type soundness for a program which contains a mixture of typed and untyped code cannot mean the traditional absence of errors. While some errors will be caught at type-checking time, other errors will only be caught as the program executes. After a decade of research there are still a number of competing approaches to providing gradual type support for object-oriented languages. We introduce a framework for comparing gradual type systems, combining a common source languages with KafKa, a core calculus for object-oriented gradual typing. KafKa decouples the semantics of gradual typing from those of the source language. KafKa is strongly typed in order to highlight where dynamic operations are required. We illustrate our approach by translating idealizations of four different gradually type approaches into the core calculus and discuss the implications of their respective designs.

1998 ACM Subject Classification F.3.3 Studies of Program Constructs

Keywords and phrases Gradual typing, object-orientation, language design, type systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.-1

1 Introduction

"Because half the problem is seeing the problem"

There has never been a single approach to gradual typing. The field was opened by two independently developed, simultaneously published papers. One, by Siek and Taha, typed individual Scheme terms using a consistency relation, inserting casts based on simple type inference [18]. The other, by Tobin-Hochtstadt and Felleisen, implemented a system allowing programmers to add types to individual modules, using constraint solving to determine where casts were needed in untyped code [25]. These two approaches set the direction for a decade of research. Today, gradual type systems support a variety of languages, enforcement mechanisms, and soundness guarantees; this degree of linguistic diversity is not without consequence, however, as the very notion of what constitutes an error remains unsettled.

The type system and semantics of a programming language are necessarily tightly coupled; each has to deal with the language's complexity. As a result, the same gradual typing semantics may seem very different when applied to two different languages, an issue that shows up clearly in object-oriented languages. Siek and Taha's first effort [19] presented a gradual type system for a variant of Abadi and Cardelli's object-based calculus [1]. It related objects by generalizing the notion of consistency [18] over structural subtyping. This early work had drawbacks, most notably in its handling of mutable state and aliasing — vital components of object-oriented languages. Underlying each subsequent gradual type system are different choices on how to deal with state and aliasing – often with key differences in soundness properties and enforcement. Our goal is to explore the design space of gradual types for objects and allow comparisons within a common framework.

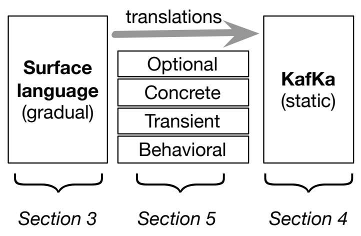
The landscape of gradually typed object-oriented languages is rich and varied. Consider some of the languages in this space:

- Typed Racket: a rich gradual type system based on contracts.
- Gradualtalk: a variant of Smalltalk with contracts.
- C#: a statically typed language with a dynamic type.
- Dart: a class based language with optional types.
- Hack: a statically typed variant of PHP that allows untyped code.
- Thorn: a language with both statically typed and untyped code.
- TypeScript: JavaScript with optional types.
- StrongScript: a variant of TypeScript with nominal types.
- Nom: a language supporting dynamic types and nominal typing.
- Reticulated Python: a family of gradual type systems for Python.

These languages can be categorized according to their runtime enforcement strategy. We can identify four major approaches, labelled here as optional, concrete, behavioral and transient. The *optional* approach, chosen by TypeScript, Dart, and Hack, amounts to static type-checking followed by type-erasure. Erroneous values flowing from dynamically typed code to statically typed code will not be caught. The *concrete* approach, used in C# and Nom, uses runtime subtype tests on type constructors to supplement static typing. While statically typed code executes at native speed, higher-order values are checked to be of statically known type at typed-untyped boundaries. The *behavioral* approach of Typed Racket, Gradualtalk and Reticulated Python monitors values to ensure that they behave in accordance to their assigned types. Instead of checking higher order values for static type tags like concrete, they add wrappers over higher order behavior to ensure it acts as the type says it should. The *transient* approach, specific to Reticulated Python, lies between concrete and behavioral; it adds type casts but does so only for the top-level of data structures. Finally, Thorn and StrongScript combine the optional and concrete semantics, differentiating between types that are erased and types that are run-time checked.

Static type systems are designed to capture some class of errors. For gradual type systems, however, it is impossible to rule out all errors; untyped code can violate type guarantees. The meaning of error for a gradual type system therefore depends on how it enforces type guarantees upon untyped code. As a result, the meaning of error depends on the choice of run-time enforcement strategy. In other words, each gradual type system may catch different "errors". We demonstrate this with a litmus test consisting of three simple programs capable of distinguishing all four major approaches to gradual typing. The litmus test programs are statically well-typed, and are "correct" in the sense that they run to completion without error in an untyped language. However, when executed with different gradual typing semantics, they produce different errors. For intuition, consider a call, x.m(). In the concrete approach, if variable x has type C (some class in the program) with method m, this statement is guaranteed to succeed. Under the behavioral semantics, the call will go through, but the function might error if it attempts to return the wrong value. In the transient semantics, the call is similarly guaranteed to go through, but might return the wrong value and not produce an error. Finally, in the optional semantics, it could error at any time, including the initial call.

We propose to compare approaches to gradual typing for objects by translating a gradually typed surface language to a target language, called KafKa. Our surface language is a *gradually* typed class-based object oriented language, similar to Featherweight Java. KafKa is a *statically* typed class-based object calculus with explicit mutable state. The key difference between the languages



is the statically sound type system and casts of KafKa. Where the surface language will allow implicit coercions, KafKa requires explicit casts to convert types. The KafKa language supports two kinds of casts: *structural casts* check if the object is a subtype of some type `t`, while *behavioral casts* create wrappers, enforcing that an object behaves as if it was of some type `t`. Translating from the gradually typed source language to the statically typed target language involves adding casts, the location and type of which depends on the gradual typing semantics.
This paper makes the following contributions:

- The design of KafKa, a common calculus for gradual type systems for objets.
- Translations from a common source to KafKa implementing each gradual approach.
- A litmus test comprised of three programs, able to tell apart the gradual type systems.
- Supplementary material includes a mechanized proof of soundness of KafKa's type system, including class generation, and a proof-of-concept implementation of KafKa on the .Net Common Language Runtime.

Our work does not address the question of performance of the translations. Each of the semantics for gradual typing has intrinsic performance costs, but these can be mitigated by compiler and runtime optimizations which we do not perform. A departure from previous work is that KafKa is statically typed. By translating to a statically typed core, we can clearly see where wrapper-induced dynamic errors can occur. Another design choice is the use of structural subtyping in KafKa. This is motivated by our desire to represent behavioral and transient approaches that require structural subtyping. There would be no major difficulty switching KafKa to a nominal subtype system or providing an additiomal nominal subtype cast. Code and proofs are available from: `github.com/BenChung/GradualComparisonArtifact`.

2 Background

"If you know the enemy and know yourself…"

The intellectual lineage of gradual typing can be traced back to attempts to add types to Smalltalk and LISP. On the Smalltalk side, work on the Strongtalk optional type system [8] led to Bracha's notion of pluggable types [7]. For him, types exist solely to catch errors at compile-time, never affecting the run-time behavior of programs. The rationale is that types exist to provide aid to the programmer and only create additional errors. In the terms of Richards *et al.* [17], an optional type system is *trace preserving*; that is to say, if a term `e` reduce to `a`, then adding type annotations to `e` does not prevent `e` from reducing. This property is valuable to developers as it ensures that type annotations will not introduce errors. Optional type systems in wide use include Hack [24], TypeScript [4] and Dart [23].

Felleisen and his students have contributed substantially to functional gradual typing. The Typed Scheme [26] design that later became Typed Racket is influenced by their earlier work on higher-order contracts [11]. Typed Racket was envisioned as a vehicle for teaching programming. Thus, being able to explain the source of errors was an important design consideration. Another consideration was to prevent surprises for beginning users. Values of type `t`, for example, should act like type `t` even when typed differently. To aid debugging, any departure from the expected behavior of an object, as defined by its behavioral type, is reported at the first discrepancy. Whenever a value crosses a boundary between typed and untyped code, it is wrapped in a contract that monitors its behavior. This ensures that mutable values remain consistent with their declared type and functions respect their declared interface. When a value misbehaves, blame can be assigned to a boundary. The granularity of typing is the module, thus a module is either entirely typed or entirely untyped. Typed Racket's support for objects was described by Takikawa et al. in [22].

Siek and Taha coined the term gradual typing in [18] as "any type system that allows programmers to control the degree of static checking for a program by choosing to annotate function parameters with types, or not." They formalized this idea in the lambda calculus augmented with references. To make the type system a gradual one, they defined the type consistency relation $\mathtt{t} \sim \mathtt{t}'$. If $\mathtt{t} \sim \mathtt{t}'$, then $\mathtt{t}$ is consistent with $\mathtt{t}'$, and can therefore be used implicitly where a $\mathtt{t}'$ instance is expected. This enables gradual typing, as $\star \sim \mathtt{t}$ for every $\mathtt{t}$ and vice versa, allowing untyped values to be passed where typed ones are expected. In [19] the authors extended this idea to an object calculus. In order to do so, they combined consistency with structural subtyping, producing consistent subtyping. With consistent subtyping, consistency can be used when checking structural subtyping, allowing typed objects and untyped objects to be passed in one another's places. However, both the basic work and its extension to classes use the same enforcement mechanism, the behavioural semantics. To explore the space of semantics further, Reticulated Python [27] is a compromise between soundness and efficiency. The language has three modes: the *guarded* mode behaves as Typed Racket with contracts applied to values. The *transient* mode performs shallow subtype checks on reads and method returns, only validating if the value obtained has matching method types. The *monotonic* mode is fundamentally different from any of the other previous approaches. Under the monotonic semantics, a cast updates the type of an object in place by replacing some of the occurrences of $\star$ with more specific types, which can then propagate recursively through the heap until a fixed point is reached.

Other noteworthy systems include Gradualtalk [2], C# 4.0 [5], Thorn [6], Nom [14] and StrongScript [17]. Gradualtalk is a variant of Smalltalk with Felleisen-style contracts and mostly nominal type equivalence (structural equivalence can be specified on demand, but it is rarely used). C# 4.0 adds the type `dynamic` to C# and dynamically resolved method invocation. Thus C# has a dynamic sublanguage that allows developers to write unchecked code, working alongside a strongly typed sublanguage in which values are guaranteed to be of their declared type. The implementation replaces $\star$ by the type `object` and adds casts where needed. Thorn and StrongScript extend the C# approach with the addition of optional types (called *like types* in Thorn). Thorn is implemented by translation to the JVM. StrongScript translate to an extended version of V8. The presence of concrete types means that the compiler can optimize code (unbox data and in-line methods) and programmers are guaranteed that type errors will not occur within concretely typed code. Nom is similar to Thorn in that it is nominal and follows the concrete approach.

Fig. 1 reviews implemented gradual type systems for objects. All languages here are class-based, except TypeScript which has both classes and plain JavaScript objects. The choice of whether to include classes is not crucial; classes are useful as a source of type declarations, but are nonessential to the type system. Most languages build subtyping on explicit subtype declarations – nominal subtyping – rather than on structural similarities. TypeScript uses structural subtyping, but does not implement a run-time check for it (as it is unsound). However, anecdotal evidence suggests that most used TypeScript type relations are declared nominally; structural relations are not widely used in TypeScript code [17]. Taking advantage of this, StrongScript uses nominal subtyping instead for improved performance in checking values, an key change due to StrongScript's dynamically checked sound types. While nominal subtyping leads to more efficient type casts, Reticulated Python's subtype consistency relation is fundamentally structural; it would be nonsensical to use it in a nominal system. For Racket, the heavy use of first-class classes and class generation naturally leads

	Nominal	*Optional*	*Concrete*	*Behavioral*	*Class based*	*First-class Class*	*Soundness claim*	*Unboxed prim.*	*Subtype cast*	*Shallow subtype cast*	*Behavioral cast*	*Blame*	*Pathologies*
Dart	●	●			●				●				-
Hack	●	●			●				●				-
TypeScript		●			●								-
C#	●		●		●		●[2]	●	●				-
Thorn	●	●	●		●		●[2]	●	●				0.8x
StrongScript	●	●	●	●	●		●[2]		●		●		1.1x
Nom	●		●		●		●[2]	●	●				1.1x
Gradualtalk	●[1]			●	●		●				●	●	5x
Typed Racket				●	●	●	●			●	●	●	121x
Reticulated Python													
Transient		●			●		●			●		●	10x
Monotonic				●	●		●				●	●	27x
Guarded				●	●		●				●	●	21x

Figure 1 Gradual type systems. (1) Opt. structural constraints. (2) Typed expressions are sound.

to structural subtyping as many of the classes being manipulated have no names and arise during computation.

The optional approach is the default for Dart, Hack and TypeScript. Transient Reticulated Python allows any value to flow in a field regardless of type annotations, leading to its "open world" soundness guarantee [27]. In Thorn, Nom and C#, primitives are concretely typed; they can be unboxed without tagging. The choice of casts follows from other design decisions. The concrete approach naturally tend to use subtype tests to establish the type of values. For nominal systems, there are highly optimized algorithms. Shallow casts are casts that only check the presence of methods, but not their signature. These are used by Racket and Python to ensure some basic form of type conformance. Behavioral casts are used when information such as a type or a blame label must be associated with a reference or an object.

Blame assignment is a topic of investigation in its own right. Anecdotal evidence suggests that the context provided by blame helps developers pinpoint the provenance of errors. In the same way that a Java stack trace identifies the function that went wrong, blame identifies where a type assertion came from. This is especially important in behavioural gradual type systems, as type assertions become wrappers which can propagate through the heap. Blame identifies where a failing wrapper came from, a task that would otherwise require extensive backtracking debugging. Unlike stack traces, which have little run-time cost, blame tracking has considerable cost due to its metadata. Blame information has to be stored whenever a wrapper is applied, a substantial overhead, and is reported to cause substantial slowdowns. However, no concrete data exists for this slowdown. We are primarily concerned with where the error arises, rather than what information is included in the error; we do not consider blame further.

The last column of Fig. 1 lists self-reported performance pathologies. These numbers are not comparable as they refer to different programs and different configurations of type annotations. They are not worst case scenarios either; most languages lack a sufficient corpus of code to conduct a thorough evaluation. Nevertheless, one can observe that for optional types no overhead is expected, as the type annotations are erased during compilation. Concrete types insert efficient casts, and lead to code that can be optimized. The performance of the transient semantics for Reticulated Python is a worst case scenario for concrete types – i.e. there is a cast at almost every call. Finally, languages with behavioral casts are prone to significant slow downs. Compiler optimizations for reducing these overheads are an active research topic [16, 3]. Languages such as C#, Nom, Thorn, and StrongScript are designed so that the performance of fully typed code is better than untyped code, and so that mixed code performs well thanks to the relatively inexpensive nominal subtype tests.

3 A Family of Gradually Typed Languages and their Litmus Test

"There is no perfection only life"

There is no single, common, notion of what constitutes an erroneous gradually typed program—a consequence of the varied enforcement strategies. The choice of enforcement strategy is reflected in the semantics of the language, which, in turn, implies that developers have to understand the details of that strategy to avoid run-time errors. This also means that it is possible to differentiate between approaches by simply observing the run-time errors that each type system produces. We propose a litmus test consisting of three programs whose execution depends on which gradual type system is in use. Each of these programs is statically well-typed and runs without error when executed with a purely dynamic semantics. However, this varies as we use different semantics for gradual typing. We start by presenting a common surface language in which we can express our programs, and then explain why the various approaches to gradual typing yield different run-time errors.

3.1 A Common Surface Language

To normalize our presentation, we use a single common source language for all four of the gradual type systems under study. The surface language is a *gradually* typed object calculus without inheritance, method overloading or explicit type cast operations. Fig. 2 gives its syntax and an extract of its static semantics. The distinctive feature of the calculus is the presence of type $\star$ – the dynamic type. A variable of type $\star$ can hold any value, an invocation of a method with receiver of type $\star$ is always statically well-typed, and an expression of type $\star$ can appear anywhere within a typed program.

This dynamic type converts our otherwise statically typed language to being gradually typed. If a program in the surface language has no appearances of $\star$ then it will be traditionally sound (not get stuck on method invocation). A program where all variables are annotated as dynamic is fully dynamic and any invocation may get stuck. Gradual typing comes into play when an expression of type $\star$ occurs as an argument to a method that expects some other type C and conversely when an argument of type C is passed to a method that expects $\star$. The static type system of the surface language allows such implicit coercions—using the convertability relation—but run- time checks may be inserted to catch potential type mismatches. We will fully formalize the semantics of this systems in future sections in terms of KafKa, a low level object calculus; here, we appeal to the readerσ intuition.

Before presenting the litmus test, some details about the type system of the surface language may prove helpful. The subtyping relation is structural with the Amber rule [9] to enable recursion. $\mathsf{M}\ \mathsf{K} \vdash \mathsf{C} <: \mathsf{D}$ holds if class C has (at least) all the methods of class D and the arguments and return types are related by subtyping in the usual contra- and co-variant way; the class table K holds definitions of all classes and M is helper for recursion. One noteworthy feature of subtyping is that the fields of objects do not play a role in deciding if classes are subtypes. Following languages like Smalltalk, fields are encapsulated and can only be accessed from within their defining object. Syntactically, field reads and writes are limited to the self-reference this.

The static type-checking rules are standard with two exceptions: Method invocation is always allowed when the receiver e is of type $\star$; therefore $\mathsf{e.m(e')}$ has type $\star$ if the argument can have type $\star$. Secondly, we introduce a notion of convertibility, which allows the mixing of static and dynamic types.

Convertibility is used when statically typed and dynamically typed terms interact. The convertibility relation, written $\mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \mathsf{t'}$, states that type t is convertible to type $\mathsf{t'}$ in class table K. The relation is used both for up-casting and for conversions of $\star$ to non-$\star$ types. $\mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \mathsf{t'}$ holds when $\mathsf{t} <: \mathsf{t'}$, this allows up-casts. The remaining two rules allow implicit conversion to and from the dynamic type. To avoid collapsing the type hierarchy, convertibility is not transitive.

It is through convertibility that our source language becomes gradual; the implicit conversions that it allows break type soundness. The semantics of the source language must therefore add dynamic checks wherever it was used. Typed values manipulated from untyped code cannot make any type assumptions about their arguments. Similarly, untyped values in typed code may return ill-typed values.

Syntax:

$$\mathsf{k} ::= \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd_1..}\ \mathsf{md_1..}\,\} \qquad \mathsf{md} ::= \mathsf{m(x\colon t)\colon t\ \{e\}} \qquad \mathsf{fd} ::= \mathsf{f\colon t} \qquad \mathsf{t} ::= \star \mid \mathsf{C}$$

$$\mathsf{e} ::= \mathsf{x} \mid \mathsf{this} \mid \mathsf{this.f} \mid \mathsf{this.f = e} \mid \mathsf{e.m(e)} \mid \textbf{new}\ \mathsf{C(e_1..)}$$

Typing expressions:

$$\frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{x : t}} \qquad \frac{\Gamma(\mathsf{this}) = \mathsf{C} \quad \mathsf{f\colon t} \in \mathsf{K(C)}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{this.f : t}} \qquad \frac{\Gamma(\mathsf{this}) = \mathsf{C} \quad \mathsf{f\colon t} \in \mathsf{K(C)} \quad \Gamma\,\mathsf{K} \vdash_s \mathsf{e : t'} \quad \mathsf{K} \vdash_s \mathsf{t'} \Rrightarrow \mathsf{t}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{this.f = e : t}} \qquad \frac{\Gamma\,\mathsf{K} \vdash_s \mathsf{e} : \star \quad \Gamma\,\mathsf{K} \vdash_s \mathsf{e' : t}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{e.m(e')} : \star}$$

$$\frac{\Gamma\,\mathsf{K} \vdash_s \mathsf{e : C} \quad \Gamma\,\mathsf{K} \vdash_s \mathsf{e' : t} \quad \mathsf{m(t_1)\colon t_2} \in \mathsf{K(C)} \quad \mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \mathsf{t_1}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{e.m(e') : t_2}} \qquad \frac{\mathsf{f_1 : t_1..} \in \mathsf{K(C)} \quad \Gamma\,\mathsf{K} \vdash_s \mathsf{e_1 : t_1'..} \quad \mathsf{K} \vdash_s \mathsf{t_1'} \Rrightarrow \mathsf{t_1..}}{\Gamma\,\mathsf{K} \vdash_s \textbf{new}\ \mathsf{C(e_1..) : C}}$$

Convertibility:

$$\frac{\emptyset\ \mathsf{K} \vdash_s \mathsf{t} <: \mathsf{t'}}{\mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \mathsf{t'}} \qquad \frac{}{\mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \star} \qquad \frac{}{\mathsf{K} \vdash_s \star \Rrightarrow \mathsf{t}}$$

Subtyping:

$$\frac{}{\mathsf{M\ K} \vdash \star <: \star} \qquad \frac{\mathsf{C} <: \mathsf{D} \in \mathsf{M}}{\mathsf{M\ K} \vdash \mathsf{C} <: \mathsf{D}} \qquad \frac{\mathsf{M'} = \mathsf{M\ C} <: \mathsf{D} \quad \mathsf{md} \in \mathsf{K(D)} \implies \mathsf{md'} \in \mathsf{K(C)}\ .\ \mathsf{M'\ K} \vdash \mathsf{md} <: \mathsf{md'}}{\mathsf{M\ K} \vdash \mathsf{C} <: \mathsf{D}}$$

$$\frac{\mathsf{M\ K} \vdash \mathsf{t_1'} <: \mathsf{t_1} \qquad \mathsf{M\ K} \vdash \mathsf{t_2} <: \mathsf{t_2'}}{\mathsf{M\ K} \vdash \mathsf{m(x\colon t_1)\colon t_2\ \{e\}} <: \mathsf{m(x\colon t_1')\colon t_2'\ \{e'\}}}$$

Figure 2 Surface language syntax and type system (extract).

3.2 Litmus

Using three different programs, we can differentiate between four gradual type systems. The litmus test is shown in Fig. 3 and its constituent programs are written in our surface language. Each of these programs consists of a class table and an expression whose evaluation in the context of the class table determines if the litmus test succeeds or fails.

The programs are designed to induce errors. This is done via the construction of an untyped value that violates the type guarantees of some systems, but not others. At heart, these programs can be summarized by the typed/untyped boundaries that are crossed by an object. Let us use the notation $\mathsf{C}\,|_{\mathsf{t}}$ to denote that an object of class C passes through a type boundary that expects it to be of type t. For example, if method m expects an argument of type t, a method call $\mathsf{e.m(e')}$ would induce the boundary

$$\mathsf{e'}\,|_{\mathsf{t}}$$

.

In program **L1**, we have

$$\mathsf{A}\,|_{\star}\,|_{\mathsf{I}}$$

: an instance of class A is first implicitly converted to $\star$ and then to I; in this program classes A and I are unrelated by subtyping. In **L2**, the same sequence of conversions is applied

$$\mathsf{A}\,|_{\star}\,|_{\mathsf{I}}$$

but this time A and I both have a method m, but the methods have incompatible argument types. Lastly, in **L3** we start by converting an object of class C to $\star$ and then to E and finally back to $\star$.

$$\mathsf{C}\,|_{\star}\,|_{\mathsf{E}}\,|_{\star}$$

the resulting value is then used in a call to method m with an instance of C as an argument. This call is correct as method m in C does take an argument of the same type. If the object was an instance of E, the call would not be legal because E's method m expects a class D as argument.
We will use this litmus test to differentiate between four different gradual type systems:
Optional. An optional gradual type system simply erases all of the type annotations at run-time; all three programs run to completion without error.

Concrete. The guarantee provided by the concrete approach is that all implicit conversion imply a run-time subtype check. This causes all three programs to fail. **L1** and **L2** fail on a subtype test $\mathsf{K} \vdash \mathsf{A} <: \mathsf{I}$. **L3** fails on the subtype test $\mathsf{K} \vdash \mathsf{C} <: \mathsf{E}$.

Behavioral. The behavioral approach does allow conversion from $\star$ to C if the actual value is “structurally” compatible to C and if, after that, the value “behaves” as if it was an instance of C. This is checked by a run-time shallow subtype cast that looks at method names, and by a wrapper that monitors further interactions. **L1** is fails at runtime because A does not have the method n expected by I. **L2**, however, executes without error because A has the method m expected by I (argument types are not checked). **L3** fails, the instance of C has been applied a wrapper that checks it behave as if it was an E. When method a is called with a C as argument, the wrapper notices that E's method a expects a D and that C and D are not compatible.

	L1	L2	L3
Concrete			
Behavioral		✓	
Transient		✓	✓
Optional	✓	✓	✓

L1

```
class A {
 m(x:A):A {this}}

class I {
 n(x:I):I {this}}

class T {
 s(x:I):T {this}
 t(x:*):* {this.s(x)}}
```

```
new T().t(new A())
```

L2

```
class A {
 m(x:A):A {this}}

class Q {
 n(x:Q):Q {this}}

class I {
 m(x:Q):I {this}}

class T {
 s(x:I):T {this}
 t(x:*):* {this.s(x)}}
```

```
new T().t(new A())
```

L3

```
class C {
 a(x:C):C {x}}

class D {
 b(x:D):D {x}}

class E {
 a(x:D):D {x}}

class F {
 m(x:E):E {x}
 n(x:*):* {this.m(x)}}
```

```
new F().n(new C())
       .a(new C())
```

Figure 3 Gradual typing litmus test.

Transient. The transient approach has a weaker guarantee. It retains the shallow structural checks at casts of the behavioral approach, but does not wrap values. Transient fails **L1**, for the same reason as the other two type systems, and passes **L2**, for the same reason as behavioral. **L3** succeeds because transient forgets that the C object was converted to E.

3.3 Discussion

The litmus tests also capture the behavior of real languages. These three programs have been expressed in TypeScript (optional), StrongScript (concrete), Typed Racket (behavioral) and Reticulated Python (transient).[1] Those implementations have the same errors.

What the litmus tests show is that understanding the implementation of gradually typed languages, the run-time enforcement machinery, is crucial; in order for developers to know when their programs is "correct," they must understand when the gradual type system will produce an error. In the litmus test, all errors are false positives, since none of these programs ever actually performs an invalid operation. This underlines the fact that in gradually typed language, false type specifications can create run-time errors just as faulty code could. Thus, type annotations must be audited and tested just like code.

The different approaches induce usability trade-offs. One way to contextualize this is with the gradual guarantee of Siek et al [20]. Put informally, the gradual guarantee states that if there exists a static type assignment to an untyped program, any partial assignment of those types will still execute successfully. Trivially, the optional approach fulfills this guarantee; under the optional approach, the types are dynamically meaningless so have no impact on the runtime. However, this comes at the cost of soundness, as the type parameters are unchecked and function calls may go wrong at any time. The transient approach likewise satisfies the gradual guarantee [28], as it only checks top-level structure of values at type boundaries. Unlike the optional approach, the transient approach does guarantee that typed function calls will succeed; however, the function may produce a dynamic error if the receiver

[1] `github.com/BenChung/GradualComparisonArtifact/examples`

is of the wrong type (even when called from a typed context), or it can return an ill-typed value to a typed call site (which is checked by the caller). The behavioral approach also fulfills the gradual guarantee, as it will only check argument and return types when wrappers are invoked. However, typed function calls might fail still if they call an untyped function that returns the wrong value. Finally, the concrete approach provides an absolute guarantee that every typed function call will be successful (e.g. method will exist, under the right argument types, and return the correct value). However, this comes at the expense of not fulfilling the gradual guarantee—partially typed classes are not compatible with more-or-less typed ones under the concrete approach.

The problem that the concrete approach has with the gradual guarantee is that the gradual guarantee is incompatible with concrete (i.e. traditional) subtyping. Suppose the fully typed version of the program relies on the static subtyping judgment $\{m(\mathsf{C}) : \mathsf{C}\} <: \{m(\mathsf{C}) : \mathsf{C}\}$. Using unaltered subtyping, no single one of these C types can be removed while retaining the subtyping relationship. To overcome this problem, Reticulated Python augments subtyping with the aforementioned *consistency* relation, or consistent subtyping [27, 19]. Consistent subtyping would admit $\{m(\star) : \mathsf{C}\} <: \{m(\mathsf{C}) : \mathsf{C}\}$, for example, as it allows consistency to be used while concluding a subtyping relation. This strictly increases the number of programs accepted by the static type system. However, consistent subtyping is not essential to the runtime semantics of any of the gradual type systems, and is fundamentally incompatible with the concrete approach (any use of consistent subtyping is guaranteed to fail). As a result, we omit consistent subtyping from our source language for the reason of simplicity.

An alternative to the gradual guarantee is presented in the approaches taken by Thorn or StrongScript. They have three kinds of types, $\star$ (dynamic), C (concrete), and like C (optional), combining the concrete and optional approach into the same language. This design allows for a different kind of migration than languages with the gradual guarantee permit; once a program is fully annotated with optional types, they all can be converted to concrete types without introducing any runtime errors [17]. We do not model this combination directly, as the underlying details are no different from the concrete approach.

The choice of structural subtyping for the surface language is motivated by the desire to support the transient approach. For the concrete approach a nominal subtyping relation is more natural (and used in Nom, Thorn and StrongScript).

The motivation for making fields private is to simplify the system. With private fields, errors are limited to method invocation. Fields accesses can be trivially checked as they are always off this. Moreover, interposing on method invocation can easily be achieved by wrappers, whereas interposing on field access would require modifying the code of clients. This would make the formal development more cumbersome without adding insight.

4 KafKa: A Core Calculus

"Aux chenilles du monde entier et aux papillons qu'elles renferment"

Even without gradual typing, comparing languages is difficult. Small differences in syntax and features can make even the most similar languages appear different. As a result, the nuances of gradual type systems are often hidden amongst irrelevant details. To enable direct comparison, we propose to translate a single gradually typed language down to a common calculus designed to highlight the distinctions between type system designs.

Our gradually typed surface language has implicit conversions; to make surface programs statically typed, we need to make these conversions into explicit casts. The target for this translation is our core language, KafKa. KafKa is a statically typed language, very similar to the source, but substituting two cast operations for the source's implicit conversions.

Additionally, KafKa makes a syntactic distinction between dynamic method invocation, written $\mathsf{e.m_{t \to t'}(e')}$, and static invocation, written $\mathsf{e@m_{\star \to \star}(e')}$. Its static type system ensures that static invocations will not go wrong, whereas dynamic invocations may get stuck. The two cast operations include a structural subtype cast, written $\langle \mathsf{t} \rangle\, \mathsf{e}$, that ensure the object that expression e evaluates to a subtype of t, and a behavioral cast, written $\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}$, that creates a wrapper around an object to monitor that it indeed behaves according to type t. The translations from the source language to KafKa precisely describe the enforcement machinery required by each approach.

KafKa was also designed to align with the capabilities of compilation target such as .NET or the JVM. It is a class-based, object-oriented, language and it supports dynamic code generation. This last feature is somewhat unusual for formal calculi but required to express behavioral casts.

4.1 Syntax and Semantics

Designing KafKa we had two requirements. First, it must be expressive enough to capture the dynamic semantics implied by each gradual type system. Second, it should have a type system that can express that some code can be statically shown to be error free. Syntax and semantics of the calculus appear in Fig. 4. They are loosely inspired by Featherweight Java [12]. At the top level, classes are notated as **class** $\mathsf{C}\,\{\, \mathsf{fd_1 .. \; md_1 ..}\,\}$, methods, ranged over by md, are denoted as $\mathsf{m(x\colon t)\colon t\ \{e\}}$, and fields $\mathsf{f\colon t}$. Expressions consist of

- variable references, x;
- self-reference this, and wrapped reference, that;
- field access, this.f, and writes, this.f = e;
- object creation, **new** $\mathsf{C(e_1 ..)}$;
- static and dynamicd method invocations;
- subtype and behavioral casts;
- references, assignment, and dereference forms for run-time use.

Evaluation is mostly standard with an evaluation context consisting of a class table K, an expression being evaluated e, and a heap σ, mapping from addresses a to objects, denoted $\mathsf{C\{a\ldots\}}$. Due to the need for dynamic code generation, the class table is part of the state. The treatment for majority of these constructs are orthogonal to the issues related to supporting gradual typing, therefore their semantics are typical for calculi of this style. Calls and casts, however, are atypical.

The static semantics hold few surprises; key typing rules appear in Fig. 4. The subtyping relation is inherited from the surface language. The program typing relation (not shown here), $\mathsf{e\ K\ \ \checkmark}$, indicates that expression e is well-formed with respect to class table K, and the expression typing judgment $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e : t}$, indicates that against Γ, with heap σ, and class table K, e has type t. Unlike the surface language, KafKa does not rely on a convertibility relation from $\star$ to C and back. Instead, explicit casts are required. The subtype cast $\langle \mathsf{t} \rangle\, \mathsf{e}$ and behavioral cast $\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}$ both allow to retype an expression e of type t′ as type t without any checks. The rule for dynamic method invocation $\mathsf{e@m_{\star \to \star}(e')}$ allows invocation of any method m on a receiver and argument of type $\star$, with a return type of $\star$.

4.1.1 Method Invocation

KafKa has two invocation forms, the dynamic $\mathsf{e@m_{\star \to \star}(e')}$ and the static $\mathsf{e.m_{t \to t'}(e')}$, both denoting a call to method m with argument e′. There are several design issues worth

Syntax:

$$
\begin{array}{lllll}
\mathsf{e} & ::= & \mathsf{x} \mid \mathsf{this} \mid \mathsf{that} \mid \\
 & & \mathsf{this.f} \mid \mathsf{this.f} = \mathsf{e} \mid \textbf{new}\ \mathsf{C}(\mathsf{e}_1..) \mid \\
 & & \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{e}) \mid \mathsf{e@m}_{\star\to\star}(\mathsf{e}) \mid \\
 & & \langle \mathsf{t}\rangle\, \mathsf{e} \mid \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} \mid \\
 & & \mathsf{a} \mid \mathsf{a.f} \mid \mathsf{a.f} = \mathsf{e}
\end{array}
\qquad
\begin{array}{lll}
\mathsf{k} & ::= & \textbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \} \\
\mathsf{md} & ::= & \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\} \\
\mathsf{fd} & ::= & \mathsf{f} : \mathsf{t} \\
\mathsf{t} & ::= & \star \mid \mathsf{C}
\end{array}
$$

Static semantics:

$$
\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C} \qquad \mathsf{m}(\mathsf{t}) : \mathsf{t}' \in \mathsf{K}(\mathsf{C}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}') : \mathsf{t}'}
\qquad
\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \star}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e@m}_{\star\to\star}(\mathsf{e}') : \star}
$$

$$
\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash \langle\mathsf{t}\rangle\, \mathsf{e} : \mathsf{t}}
\qquad
\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} : \mathsf{t}}
\qquad
\frac{\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a'}_1..\}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}}
\qquad
\frac{}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \star}
$$

Execution contexts:

$$
\begin{array}{lllllll}
\mathsf{E} & ::= & \mathsf{a.f} = \mathsf{E} & \mid \mathsf{E.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{e}) & \mid \mathsf{a.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{E}) & \mid \mathsf{E@m}_{\star\to\star}(\mathsf{e}) & \mid \\
 & & \mathsf{a@m}_{\star\to\star}(\mathsf{E}) & \mid \langle\mathsf{t}\rangle\, \mathsf{E} & \mid \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{E} & \mid \textbf{new}\ \mathsf{C}(\mathsf{a}_1..\,\mathsf{E}\,\mathsf{e}_1..) & \mid \square
\end{array}
$$

Dynamic semantics:

$$
\begin{array}{llllllllll}
\mathsf{K} & \textbf{new}\ \mathsf{C}(\mathsf{a}_1..) & \sigma & \to & \mathsf{K} & \mathsf{a}' & \sigma' & \textbf{where} & \mathsf{a}'\ \mathsf{fresh} \qquad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\mathsf{a}_1..\}] \\
\mathsf{K} & \mathsf{a.f}_i & \sigma & \to & \mathsf{K} & \mathsf{a}_i & \sigma & \textbf{where} & \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}_i, \mathsf{a}_n \ldots\} \\
\mathsf{K} & \mathsf{a.f}_i = \mathsf{a}' & \sigma & \to & \mathsf{K} & \mathsf{a}' & \sigma' & \textbf{where} & \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}_i, \mathsf{a}_n \ldots\} \\
 & & & & & & & & \sigma' = \sigma[\mathsf{a} \mapsto \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}', \mathsf{a}_n \ldots\}] \\
\mathsf{K} & \mathsf{a.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{a}') & \sigma & \to & \mathsf{K} & \mathsf{e}' & \sigma & \textbf{where} & \mathsf{e}' = [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e} \\
 & & & & & & & & \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1)\colon \mathsf{t}_2\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \\
 & & & & & & & & \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \qquad \emptyset\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}_1 \\
 & & & & & & & & \emptyset\ \mathsf{K} \vdash \mathsf{t}_2 <: \mathsf{t}' \\
\mathsf{K} & \mathsf{a@m}_{\star\to\star}(\mathsf{a}') & \sigma & \to & \mathsf{K} & \mathsf{e}' & \sigma & \textbf{where} & \mathsf{e}' = [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e} \\
 & & & & & & & & \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \\
 & & & & & & & & \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \\
\mathsf{K} & \langle\star\rangle\, \mathsf{a} & \sigma & \to & \mathsf{K} & \mathsf{a} & \sigma & & \\
\mathsf{K} & \langle\mathsf{D}\rangle\, \mathsf{a} & \sigma & \to & \mathsf{K} & \mathsf{a} & \sigma & \textbf{where} & \emptyset\ \mathsf{K} \vdash \mathsf{C} <: \mathsf{D} \qquad \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \\
\mathsf{K} & \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{a} & \sigma & \to & \mathsf{K}' & \mathsf{a}' & \sigma' & \textbf{where} & \mathsf{K}'\ \mathsf{a}'\ \sigma' = \mathsf{bcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) \\
\mathsf{K} & \mathsf{E}[\mathsf{e}] & \sigma & \to & \mathsf{K}' & \mathsf{E}[\mathsf{e}'] & \sigma' & \textbf{where} & \mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'
\end{array}
$$

Figure 4 KafKa dynamic semantics and static semantics (extract).

discussing. First, as our calculus is a translation target, it is acceptable to require some explicit preparation for objects to be used in a dynamic context. A dynamic call is only successful if the receiver has a method of the expected name and argument and return types of $\star$. Thus, even dynamic invocation has to be well-typed. Secondly, it is possible for a static invocation to call an untyped method (of type $\star$ to $\star$).

To examine KafKa's semantics for method calls, consider the following class definition

```
class C {
        m(x: Int): Int {  x + 2  }
```

```
    m (x: ⋆) : ⋆ { ⟨⋆⟩ this.m_{Int→Int}(⟨Int⟩ x) }
}
```

and assume that class table K holds a definition for Int. Furthermore we take liberties with syntax around addition and integers for illustration.

The class above demonstrates several features of KafKa. Its class well-formedness rules (not shown here) allow a limited form of method overloading. A class may have at most two occurences of any method m. One, which we call "untyped", with $\star$ as argument and return type. And one, which call "typed", with either argument and return type differ from $\star$. The static type system enforces a single means of invoking a typed method m:

new $\mathsf{C}().\mathsf{m}_{\mathsf{Int}\to\mathsf{Int}}(2)$

Here the receiver is obviously of type C and the argument is Int, thus the call is statically well-typed. The expression is thus guaranteed to evaluate the body of m. For an untyped method, there are two invocation modes:

new $\mathsf{C}()@\mathsf{m}_{\star\to\star}(2)$

in a dynamic invocation, the method may be missing, and thus evaluation may get stuck. On the other hand, one could use static method invocation form such as

new $\mathsf{C}().\mathsf{m}_{\star\to\star}(2)$

then the receiver is checked for an untyped method, and the invocation is guaranteed to succeed. All nuances will come in handy when translating the surface language to KafKa.

4.1.2 Run-time Casts

KafKa has two cast operations: the subtype cast $\langle \mathsf{t}\rangle\, \mathsf{e}$ and the behavioral cast $\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}$, both indicating the desire that the result of evaluating e has type t. Where the casts differ is what is meant by "has type t". The subtype casts checks that the result of evaluating e is an object whose class is a subtype of t. If $\mathsf{t} = \star$ then the cast trivially succeeds. This cast is possible because every value in the heap is tagged by its type construct. The behavioral cast is more complex, we will describe it in the remainder.

The behavioral cast wraps the result of evaluating e in a newly created object which enforces the invariant that the object *behaves* as a value of type t. Function $\mathsf{bcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) = \mathsf{K}'\, \mathsf{a}'\, \sigma'$ specifies its semantics, shown in Fig. 5. There are two cases to consider, either the target type is a class C', or it is $\star$.

If the target type is C', then $\mathsf{bcast}(\mathsf{a}, \mathsf{C}', \sigma, \mathsf{K})$ will return an updated class table K', a reference to the wrapped object a', and an updated heap σ'. The only property immediately checked is that a has all the method names declared by C'. If this is not the case, it is impossible for the object to implement the desired interface, and an early failure is acceptable. The wrap function generates a wrapper, that is, it generates code for a new class for the wrapper object, instantiates it, and returns a wrapper a' that points to a. The metafunction W does the heavy lifting of class creation. It is used as follows, $\mathsf{W}(\mathsf{C}, \mathsf{md}_1.., \mathsf{md}'_1.., \mathsf{D})$ takes a class C, a fresh name D, and two method lists $\mathsf{md}_1..$ and md'_1, respectively the method of C and the methods of the type to enforce. The generated class will have adapter methods for each method m occurring in both $\mathsf{md}_1..$ and md'_1... The adapter will introduce behavioral casts for argument and return values. For method that do not need to be adapted (methods only in $\mathsf{md}_1..$) as simple pass-through method is generated that calls the wrapped object; that object is referenced by distinguished variable that.

Behavioral cast: $\mathsf{bcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) = \mathsf{K}'\ \mathsf{a}'\ \sigma'$

a	Reference to wrap	a'	Wrapped reference
t	Target type to enforce	K'	Class table with wrapper
σ	Original heap	σ'	New heap
K	Original class table		

$$\mathsf{bcast}(\mathsf{a}, \mathsf{C}', \sigma, \mathsf{K}) = \mathsf{K}'\ \mathsf{a}'\ \sigma' \quad \textbf{where} \quad \begin{cases} \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \quad \mathsf{D}, \mathsf{a}' \text{ fresh} \quad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}] \\ \mathsf{md}_1.. \in \mathsf{K}(\mathsf{C}) \quad \mathsf{names}(\mathsf{md}'_1..) \subseteq \mathsf{names}(\mathsf{md}_1..) \\ \mathsf{md}'_1.. \in \mathsf{K}(\mathsf{C}') \quad \mathsf{nodups}(\mathsf{md}_1..) \ \ \mathsf{nodups}(\mathsf{md}'_1..) \\ \mathsf{K}' = \mathsf{K}\ \mathsf{W}(\mathsf{C}, \mathsf{md}_1.., \mathsf{md}'_1.., \mathsf{D}) \end{cases}$$

$$\mathsf{bcast}(\mathsf{a}, \star, \sigma, \mathsf{K}) = \mathsf{K}'\ \mathsf{a}'\ \sigma' \quad \textbf{where} \quad \begin{cases} \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \quad \mathsf{md}_1.. \in \mathsf{K}(\mathsf{C}) \quad \mathsf{D}, \mathsf{a}' \text{ fresh} \\ \mathsf{nodups}(\mathsf{md}_1..) \quad \mathsf{K}' = \mathsf{K}\ \mathsf{W}\star(\mathsf{C}, \mathsf{md}_1.., \mathsf{D}) \\ \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}] \end{cases}$$

$$\begin{aligned} \mathsf{W}(\mathsf{C}, \mathsf{md}_1.., \mathsf{md}'_1.., \mathsf{D}) &= \textbf{class } \mathsf{D}\,\{\,\mathsf{that}\colon \mathsf{C}\ \mathsf{md}''_1..\,\} \\ \textbf{where}\quad & \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1)\colon \mathsf{t}_2\ \{\mathsf{e}\} \in \mathsf{md}_1.. \\ \mathsf{md}''_1 &= \mathsf{m}(\mathsf{x}\colon \mathsf{t}'_1)\colon \mathsf{t}'_2\ \{\ \blacktriangleleft \mathsf{t}'_2 \blacktriangleright \mathsf{this.that.m}_{\mathsf{t}_1 \to \mathsf{t}_2}(\blacktriangleleft \mathsf{t}'_1 \blacktriangleright \mathsf{x})\}\ .. \\ & \quad \textbf{if}\quad \mathsf{m}(\mathsf{x}\colon \mathsf{t}'_1)\colon \mathsf{t}'_2\ \{\mathsf{e}'\} \in \mathsf{md}'_1.. \\ & \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1)\colon \mathsf{t}_2\ \{\ \mathsf{this.that.m}_{\mathsf{t}_1 \to \mathsf{t}_2}(\mathsf{x})\}\ .. \\ & \quad \textbf{otherwise} \\ \mathsf{W}\star(\mathsf{C}, \mathsf{md}_1.., \mathsf{D}) &= \textbf{class } \mathsf{D}\,\{\,\mathsf{that}\colon \mathsf{C}\ \mathsf{md}'_1..\,\} \\ \textbf{where}\quad \mathsf{md}'_1 &= \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{this.that.m}_{\mathsf{t} \to \mathsf{t}'}(\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x})\}\ .. \\ & \quad \textbf{if}\quad \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\mathsf{e}\} \in \mathsf{md}_1.. \end{aligned}$$

Figure 5 Behavioral cast semantics.

If the target type is $\star$, the wrapper class is simpler. It only needs to check that method arguments matches the type expected by the wrapped object. This is done by another behavioral cast. Return values are cast to $\star$.

For example, consider the following program which has two classes C ad D. Even though C and D both have method a, they are not subtypes because the argument of m are not related.

$$(\blacktriangleleft \mathsf{C} \blacktriangleright (\blacktriangleleft \mathsf{D} \blacktriangleright \textbf{new } \mathsf{C}())).\mathsf{b}(2) \qquad \textbf{where} \quad \mathsf{K} = \begin{array}{l} \textbf{class } \mathsf{C}\ \{ \\ \quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \mathsf{x}\ \} \\ \quad \mathsf{n}(\mathsf{x}\colon \star)\colon \star\ \{\ \mathsf{x}\ \} \\ \} \\ \textbf{class } \mathsf{D}\ \{\ \ \mathsf{m}(\mathsf{x}\colon \mathsf{Int})\colon \mathsf{Int}\ \{\ \mathsf{x}\ \}\ \ \} \end{array}$$

The program starts with a C, casts it to D, and then back to C. The reason we generate pass-through methods (the wrapper that enforce type D has a method n) is that without them, the methods would be “lost”. In When we cast the object back to C, it has two layers of wrapper and the pass-through method is needed to be able to invoke n.

4.2 Type soundness

The KafKa type soundness theorem ensures that a well-formed program can only get stuck at a dynamic invocation, a subtype cast, or a behavioral cast, and only there when justified.

▶ **Theorem 1** (KafKa type soundness). *For every well-formed-state* $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ *and well-typed expression* $\emptyset\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$*, one of the following holds:*

- *There exists some reference* a *such that* $\mathsf{e} = \mathsf{a}$*.*
- $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$*, where* $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$*,* $\emptyset\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$*,* σ' *has all of the values of* σ*, and* K'*has all of the classes of* K*.*
- $\mathsf{e} = E[\mathsf{a}@\mathsf{m}_{\star\rightarrow\star}(\mathsf{a}')]$ *and* a *refers to an object without a method* m*.*
- $\mathsf{e} = E[\langle \mathsf{C} \rangle\ \mathsf{a}]$*, and* a *refers to an object whose class is not a subtype of* C*.*
- $\mathsf{e} = E[\blacktriangleleft \mathsf{C} \blacktriangleright\ \mathsf{a}]$*, and* C *contains a method that* a *does not.*

The proof is mostly straightforward, with one unusual case, centered around the bcast metafunction. When the bcast metafunction is used to generate a wrapper class, which is then instantiated, producing a new class table and heap, we must then show that the new class table is well formed, that the new heap is also well formed, and that the new wrapper is a subtype of the given type C. Proving these properties is relatively easy. Class table well-formedness follows by construction of the wrapper class and by well-formedness of the old class table. Heap well-formedness follows by well-formedness of the class table, construction of the new heap, and well-formedness of the old heap. Proving that the type of the wrapper is a subtype of the required type proceeds by structural induction over the required type.

The proof of soundness has been formalized in Coq; available in the supplementary material. The proof has two axioms: recursive structural subtyping is transitive and correct. We did not prove these (which have been shown in prior work [13]) as they lie outside the scope of our work.

4.3 Discussion

The design of KafKa's two invocation forms bears discussion. In some previous works, dynamic invocation has been implemented by a combination of a cast and a statically typed call. In our case, following this approach would require creating a type for each invocation (as was done in [29]), providing a dynamic invocation form seemed more natural and simpler. The use of explicitly typed invocation is a result of our desire to be able to rule out some errors statically. The particular design comes from the observation that the same method name needs to called from both typed and untyped contexts. Depending on the gradual semantics, calls from a typed or untyped context may require different attention, we decided to all classes to have two versions of the method and add types to the method name to disambiguate between them.

We intended KafKa to match the intermediate languages of commercial VMs. To validate this, we implemented a compiler from KafKa to C#.[2] The only challenge was due to subtyping. KafKa uses structural typing, while C# is nominal, and KafKa allows methods in subtypes to be contra-variant in argument and co-variant in return type, while C# requires invariance. Implementing structural subtyping on top of a nominally typed language is tricky. Structural types create implicit subtyping relationships, which the nominal type system expects to be

[2] `github.com/BenChung/GradualComparisonArtifact/netImpl`

explicit. Prior work used reflection and complex run-time code generation [10], but this is needlessly complex for a proof of concept. Instead, we reify the implicit relationships introduced by structural subtyping into explicit nominal relationships by generating interfaces. Given two classes C and D, where $\mathsf{K} \vdash \mathsf{C} <: \mathsf{D}$ holds, we generate two interfaces CI and DI, where CI is declared to extend DI. As a result, if two types are subtypes, their corresponding C# interfaces will be as well. The next problem is that KafKa allows subtype methods to be contra-variant in argument and co-variant in return types. As a result, a single methods in CI may not be sufficient to implement DI. We solve this by having every class's C# equivalent implement every interface explicitly, with each explicit implementation delegating to the real, most general, implementation. Despite these issues, we were able to accurately translate KafKa types. We translate static and dynamic invocations into corresponding C# invocations since C# has also a dynamic type. The underlying run-time can then use the translated KafKa types to perform method dispatch, while inserting dynamic checks wherever the KafKa source calls for an untyped invocation. This prototype shows that KafKa primitives are close to those of intermediate languages. As a result, the translation of gradual type systems to KafKa provides insight as to how they might be implemented in a practical language.

5 Translating Gradual Type Systems

“Was ist mit mir geschehen? dachte er. Es war kein Traum”

The four gradual type semantics are now ready to be translated into KafKa. Each semantics is translated through a function mapping well-typed surface programs into well-typed KafKa terms. The translation explicitly determines which type casts need to be inserted and the invocation forms to use. The surface languages have no explicit casts, instead they coerce types at typed-untyped boundaries. A type-driven translation will insert the needed casts. Consider the following example where method m takes an argument of type $\star$ and returns it under type C. If m is passed a D (which has no subtype relation with C), the returned value will not match the declared return type. Without a cast, this operation is unsafe.

$$\textbf{new}\ \mathsf{C}().\mathsf{m}(\textbf{new}\ \mathsf{D}()) \qquad \textbf{where}\quad \mathsf{K} = \ \textbf{class}\ \mathsf{D}\ \{\} \quad \textbf{class}\ \mathsf{C}\ \{\ \mathsf{m}(\mathsf{x}\colon \star)\colon \mathsf{C}\ \{\ \mathsf{x}\ \}\ \}$$

5.1 Class Translation

The translations for source level classes are shown in Fig. 6. Each class in the surface language translates to a homonymous KafKa class, thus type names are retained in the translation. Grey background denotes generated code.

Optional. The optional approach provides no correctness guarantees. Retaining the surface type annotations through translation would not preserve this semantics, so we erase them. The resulting class has all fields as well all method arguments and return values typed as $\star$.

Transient. The transient approach guarantees the presence of methods, but not their signature. Since fields are not part of types, they will not be checked. The translation sets them all to $\star$. Methods are translated to accept $\star$ and return $\star$. For a method m that accepted a value x of type t in the surface language, the translation will first check that x is of type by the means of subtype cast. The subtype cast operates on the translated type t–the type with all methods untyped.[3]

[3] We abuse the notation $\mathsf{e}; \mathsf{e}'$ to denote sequence, this could be encoded in the calculus, but explicit sequence is easier to read.

Optional:

$$\begin{array}{lll}
\mathcal{O}[\![\mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] & = & \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}'_1..\ \mathsf{md}'_1..\,\} \\
& \mathbf{where} & \mathsf{fd}'_1 = \mathsf{f}:\star\ ..\quad \mathsf{fd}_1 = \mathsf{f}:\mathsf{t}.. \\
& & \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\star):\star\ \{\mathsf{e}'\}\ .. \\
& & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}_1):\mathsf{t}_2\ \{\mathsf{e}\}\quad \mathsf{e}' = \mathcal{O}[\![\mathsf{e}]\!]
\end{array}$$

Transient:

$$\begin{array}{lll}
\mathcal{T}[\![\mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] & = & \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}'_1..\ \mathsf{md}'_1..\,\} \\
& \mathbf{where} & \mathsf{fd}'_1 = \mathsf{f}:\star\ ..\quad \mathsf{fd}_1 = \mathsf{f}:\mathsf{t}.. \\
& & \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\star):\star\ \{\langle\mathsf{t}\rangle\,\mathsf{x}\ ;\ \mathsf{e}'_1\}\ .. \\
& & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}):\mathsf{t}'\ \{\mathsf{e}\}..\quad \mathsf{e}'_1 = \mathcal{T}(\!|\mathsf{e}|\!)^{\star}_{\mathsf{x}:\mathsf{t}\ \mathsf{this}:\mathsf{C}}\ ..
\end{array}$$

Behavioral:

$$\begin{array}{lll}
\mathcal{B}[\![\mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] & = & \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}'_1..\,\} \\
& \mathbf{where} & \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}):\mathsf{t}'\ \{\mathsf{e}'_1\}\ .. \\
& & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}):\mathsf{t}'\ \{\mathsf{e}_1\}\ ..\quad \mathsf{e}'_1 = \mathcal{B}[\![\mathsf{e}_1]\!]_{\mathsf{x}:\mathsf{t}\ \mathsf{this}:\mathsf{C}}
\end{array}$$

Concrete:

$$\begin{array}{lll}
\mathcal{C}[\![\mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] & = & \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}'_1..\mathsf{md}''_1..\,\} \\
& \mathbf{where} & \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}_1):\mathsf{t}_2\ \{\mathsf{e}'\}\ .. \\
& & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}_1):\mathsf{t}_2\ \{\mathsf{e}\}..\quad \mathsf{e}' = \mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}_2}_{\mathsf{this}:\mathsf{C}\ \mathsf{x}:\mathsf{t}_1}\ .. \\
& & \mathsf{md}''_1 = \mathsf{m}(\mathsf{x}:\star):\star\ \{\langle\star\rangle\,\mathsf{this}.\mathsf{m}_{\mathsf{t}_1\to\mathsf{t}_2}(\langle\mathsf{t}_1\rangle\,\mathsf{x})\} \\
& & \qquad\qquad \mathbf{if}\ \ \mathsf{t}_1 \neq \star \\
& & \qquad \mathit{empty}\ \ \mathbf{otherwise}\ ..
\end{array}$$

Figure 6 Translations for classes.

Behavioral. The behavioral approach guarantees soundness by wrapping values that cross type-untyped boundaries. Methods are presevered by the translation but bodies are translated.

Concrete. The concrete approach ensures that variables of non-$\star$ types refer to subtypes of the given type. Each method appearing in the original class is retained as such with its body translated. Moreover, all typed method, to be called from an untyped context, need to have an untyped variant that performs subtype casts of the argument to its expected type and re-dispatches to the corresponding typed method.

5.2 Expression Translation

To accommodate differences between the gradual typing semantics, we use two different expression translation schemes. The first is a type-ambivalent one, used for the optional approach, while the second is type-aware, used for the three other approaches. $\mathcal{O}[\![\mathsf{e}]\!]$ denotes optional translation, where e is the target expression, and the result is a KafKa term. The type-aware translation has two forms, $\mathcal{S}[\![\mathsf{e}]\!]_\Gamma$ and $\mathcal{S}(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma$, inspired by work on bidirectional type-checking [15]. The differentiation between them arises from the need to insert casts only where required. The first form, $\mathcal{S}[\![\mathsf{e}]\!]_\Gamma$, is analogous to the synthetic case in bidirectional type-checking. It is used for expressions without any specific required type. The second form,

Transient:

$$\begin{array}{lll}
\mathcal{T}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t} \quad \mathsf{e}' = \mathcal{T}[\![\mathsf{e}]\!]_{\Gamma} \\
\mathcal{T}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \langle \mathsf{t} \rangle\, \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' \not<: \mathsf{t} \quad \mathsf{e}' = \mathcal{T}[\![\mathsf{e}]\!]_{\Gamma}
\end{array}$$

Behavioral:

$$\begin{array}{lll}
\mathcal{B}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t} \quad \mathsf{e}' = \mathcal{B}[\![\mathsf{e}]\!]_{\Gamma} \\
\mathcal{B}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' \not<: \mathsf{t} \quad \mathsf{e}' = \mathcal{B}[\![\mathsf{e}]\!]_{\Gamma}
\end{array}$$

Concrete:

$$\begin{array}{lll}
\mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t} \quad \mathsf{e}' = \mathcal{C}[\![\mathsf{e}]\!]_{\Gamma} \\
\mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma} & = \langle \mathsf{t} \rangle\, \mathsf{e}' & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t}' \not<: \mathsf{t} \quad \mathsf{e}' = \mathcal{C}[\![\mathsf{e}]\!]_{\Gamma}
\end{array}$$

Figure 7 Assertive translation.

$\mathcal{S}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma}$, is used when e must have some type t. Analogous to the analytic case of bidirectional type-checking, this form applies when some enclosing expression has an expectation of the type of e. For example, it is used in translation of method arguments, which must conform to the types of the arguments to the method. We refer to this case as assertive translation.

The assertive translation of Fig. 7 is responsible for producing well-typed terms by adding casts into expressions where static types differ. The rules closely track the convertibility relation of the surface language. Every type-driven translation has two cases. The first case is used when the required type happens to be a supertype of the expression's actual type, then no further action is required. The second case handles typed-untyped boundaries, conversions to or from $\star$. The concrete and transient approaches use the subtype cast operator for dynamic type checks. The behavioral approach instead inserts behavioral casts at boundaries. For transient, since the class translation erase all types, the subtype cast really only checks the presence of method names.

The translation of field access appears in Fig. 8. The optional translation only inserts a cast to $\star$ in front of uses of the this reference as a technicality required for statically typing terms. The transient translation adds casts to the expected type of variable and fields. Again, these casts only check the presence of fields. The behavioral translation and the concrete translation leave access intact.

Optional:

$$\begin{array}{ll}
\mathcal{O}[\![\mathsf{x}]\!] & = \mathsf{x} \\
\mathcal{O}[\![\mathsf{this}]\!] & = \langle \star \rangle\, \mathsf{this} \\
\mathcal{O}[\![\mathsf{this.f}]\!] & = \mathsf{this.f}
\end{array}$$

Transient:

$$\begin{array}{lll}
\mathcal{T}[\![\mathsf{x}]\!]_{\Gamma} & = \langle \mathsf{t} \rangle\, \mathsf{x} & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{x} : \mathsf{t} \\
\mathcal{T}[\![\mathsf{this}]\!]_{\Gamma} & = \mathsf{this} & \\
\mathcal{T}[\![\mathsf{this.f}]\!]_{\Gamma} & = \langle \mathsf{t} \rangle\, \mathsf{this.f} & \textbf{where}\ \ \mathsf{K},\Gamma \vdash \mathsf{this} : \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C})
\end{array}$$

Behavioral:

$$\begin{array}{ll}
\mathcal{B}[\![\mathsf{x}]\!]_{\Gamma} & = \mathsf{x} \\
\mathcal{B}[\![\mathsf{this}]\!]_{\Gamma} & = \mathsf{this} \\
\mathcal{B}[\![\mathsf{this.f}]\!]_{\Gamma} & = \mathsf{this.f}
\end{array}$$

Concrete:

$$\begin{array}{ll}
\mathcal{C}[\![\mathsf{x}]\!]_{\Gamma} & = \mathsf{x} \\
\mathcal{C}[\![\mathsf{this}]\!]_{\Gamma} & = \mathsf{this} \\
\mathcal{C}[\![\mathsf{this.f}]\!]_{\Gamma} & = \mathsf{this.f}
\end{array}$$

Figure 8 Translations variables and field access.

The translation for assignment is shown in Fig. 9. All the approaches translate the value only differing in the expected type. Behavioral and concrete require that the result has the statically known type, transient expects $\star$, and the optional semantics imposes no type requirement whatsoever.

Optional:

$$\mathcal{O}[\![\mathsf{this.f} = \mathsf{e}]\!] \quad = \quad \mathsf{this.f} = \mathsf{e}' \qquad \textbf{where} \quad \mathsf{e}' = \mathcal{O}[\![\mathsf{e}]\!]$$

Transient:

$$\mathcal{T}[\![\mathsf{this.f} = \mathsf{e}]\!]_\Gamma \quad = \quad \mathsf{this.f} = \mathsf{e}' \qquad \textbf{where} \quad \mathsf{K}, \Gamma \vdash \mathsf{this} : \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}' = \mathcal{T}(\!|\mathsf{e}|\!)^{\star}_{\Gamma}$$

Behavioral:

$$\mathcal{B}[\![\mathsf{this.f} = \mathsf{e}]\!]_\Gamma \quad = \quad \mathsf{this.f} = \mathsf{e}' \qquad \textbf{where} \quad \mathsf{K}, \Gamma \vdash \mathsf{this} : \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}' = \mathcal{B}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma}$$

Concrete:

$$\mathcal{C}[\![\mathsf{this.f} = \mathsf{e}]\!]_\Gamma \quad = \quad \mathsf{this.f} = \mathsf{e}' \qquad \textbf{where} \quad \mathsf{K}, \Gamma \vdash \mathsf{this} : \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}' = \mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}}_{\Gamma}$$

Figure 9 Translations for assignment.

The translation for object creation, shown in Fig. 10, follows the same reasoning. It translates each argument to be the required type according to class translation.

Optional:

$$\mathcal{O}[\![\textbf{new}\ \mathsf{C}(\mathsf{e}_1..)]\!] \quad = \quad \langle\star\rangle\ \textbf{new}\ \mathsf{C}(\mathsf{e}'_1..) \qquad \textbf{where} \quad \mathsf{e}'_1 = \mathcal{O}[\![\mathsf{e}_1]\!]\ ..$$

Transient:

$$\mathcal{T}[\![\textbf{new}\ \mathsf{C}(\mathsf{e}_1..)]\!]_\Gamma \quad = \quad \textbf{new}\ \mathsf{C}(\mathsf{e}'_1..) \qquad \textbf{where} \quad \mathsf{f}_1 : \mathsf{t}_1 \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}'_1 = \mathcal{T}(\!|\mathsf{e}_1|\!)^{\star}_{\Gamma}\ ..$$

Behavioral:

$$\mathcal{B}[\![\textbf{new}\ \mathsf{C}(\mathsf{e}_1..)]\!]_\Gamma \quad = \quad \textbf{new}\ \mathsf{C}(\mathsf{e}'_1..) \qquad \textbf{where} \quad \mathsf{f}_1 : \mathsf{t}_1 \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}'_1 = \mathcal{B}(\!|\mathsf{e}_1|\!)^{\mathsf{t}_1}_{\Gamma}\ ..$$

Concrete:

$$\mathcal{C}[\![\textbf{new}\ \mathsf{C}(\mathsf{e}_1..)]\!]_\Gamma \quad = \quad \textbf{new}\ \mathsf{C}(\mathsf{e}'_1..) \qquad \textbf{where} \quad \mathsf{f}_1 : \mathsf{t}_1 \in \mathsf{K}(\mathsf{C}) \quad \mathsf{e}'_1 = \mathcal{C}(\!|\mathsf{e}_1|\!)^{\mathsf{t}_1}_{\Gamma}\ ..$$

Figure 10 Translations for object creation.

The translations for invocation are shown in Fig. 11. Optional translates all invocations to dynamic invocation. At invocation sites, a method of the right name must be known to exist with the correct argument and return types. In the concrete and behavioral semantics source typing is retained, so the arguments must be of the statically known type. However, in the transient semantics, the argument type is ignored, so the argument to a statically typed method call is only required to be of type $\star$, but the return type is checked. If any of the systems, if the type of the receiver is $\star$, dynamic invocation is used.

Optional:

$$\mathcal{O}[\![\mathsf{e_1.m(e_2)}]\!] = \mathsf{e_1'@m_{\star\to\star}(e_2')} \qquad \textbf{where} \;\; \mathsf{e_1'} = \mathcal{O}[\![\mathsf{e_1}]\!] \;\; \mathsf{e_2'} = \mathcal{O}[\![\mathsf{e_2}]\!]$$

Transient:

$$\mathcal{T}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \mathsf{e_1'@m_{\star\to\star}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \star \;\; \mathsf{e_1'} = \mathcal{T}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{T}(\!|\mathsf{e_2}|\!)^\star_\Gamma$$

$$\mathcal{T}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \langle \mathsf{D_2}\rangle\, \mathsf{e_1'.m_{\star\to\star}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \mathsf{C} \;\; \mathsf{e_1'} = \mathcal{T}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{T}(\!|\mathsf{e_2}|\!)^\star_\Gamma \;\; \mathsf{m(D_1) : D_2} \in \mathsf{K(C)}$$

Behavioral:

$$\mathcal{B}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \mathsf{e_1'@m_{\star\to\star}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \star \;\; \mathsf{e_1'} = \mathcal{B}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{B}(\!|\mathsf{e_2}|\!)^\star_\Gamma$$

$$\mathcal{B}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \mathsf{e_1'.m_{D_1\to D_2}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \mathsf{C} \;\; \mathsf{e_1'} = \mathcal{B}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{B}(\!|\mathsf{e_2}|\!)^{\mathsf{D_1}}_\Gamma \;\; \mathsf{m(D_1) : D_2} \in \mathsf{K(C)}$$

Concrete:

$$\mathcal{C}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \mathsf{e_1'@m_{\star\to\star}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \star \;\; \mathsf{e_1'} = \mathcal{C}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{C}(\!|\mathsf{e_2}|\!)^\star_\Gamma$$

$$\mathcal{C}[\![\mathsf{e_1.m(e_2)}]\!]_\Gamma = \mathsf{e_1'.m_{D_1\to D_2}(e_2')} \qquad \textbf{where} \;\; \mathsf{K},\Gamma \vdash \mathsf{e_1} : \mathsf{C} \;\; \mathsf{e_1'} = \mathcal{C}[\![\mathsf{e_1}]\!]_\Gamma \;\; \mathsf{e_2'} = \mathcal{C}(\!|\mathsf{e_2}|\!)^{\mathsf{D_1}}_\Gamma \;\; \mathsf{m(D_1) : D_2} \in \mathsf{K(C)}$$

Figure 11 Translations for function invocation.

5.3 Example

We illustrate the translation with the behavior of litmus program **L3**. The operational principle of **L3** is that it creates a new object (an instance of C), then uses an untyped intermediate to represent it as type E. Type E ascribes the wrong type for argument x, substituting D for the correct type C. However, since the object never uses its argument, this faulty type is not exercised.

Source:

$$\mathsf{class\ F\ \{\ \ m(x\colon E)\colon E\ \{x\}\quad n(x\colon \star)\colon \star\ \{this.m(x)\}\ \}}$$

Optional:

$$\mathsf{class\ F\ \{\ \ m(x\colon \star)\colon \star\ \{x\}\quad n(x\colon \star)\colon \star\ \{(\langle\star\rangle\ this)@m_{\star\to\star}(x)\}\ \}}$$

Transient:

$$\mathsf{class\ F\ \{\ \ m(x\colon \star)\colon \star\ \{\langle E\rangle\, x;\ \ \langle\star\rangle\, x\}\quad n(x\colon \star)\colon \star\ \{\langle\star\rangle\, x;\ \ \langle\star\rangle\, \langle E\rangle\ this.m_{\star\to\star}(\langle\star\rangle\, \langle\star\rangle\, x)\}\ \}}$$

Behavioral:

$$\mathsf{class\ F\ \{\ \ m(x\colon E)\colon E\ \{x\}\quad n(x\colon \star)\colon \star\ \{\blacktriangleleft\star\blacktriangleright\ this.m_{E\to E}(\blacktriangleleft E\blacktriangleright\ x)\}\ \}}$$

Concrete:

$$\mathsf{class\ F\ \{\ \ m(x\colon E)\colon E\ \{x\}\quad n(x\colon \star)\colon \star\ \{\langle\star\rangle\ this.m_{E\to E}(\langle E\rangle\, x)\}\ \}}$$

Figure 12 Class translation for litmus test L3.

Two of the gradual type systems notice this invalid type. Concrete errors on **L3** because E is not a subtype of C. With behavioral, the unused type ascription is saved as a wrapper and is enforced causing a runtime error. While this reasoning provides an intuition, it provides few detail for which we turn to our formalism. We present the translation from the top, starting with classes in Fig. 12. The optional approach does no checking whatsoever, and simply erases types. Transient also erases types, but adds argument casts on method entry. In the case of m, argument x is checked to be of type E, as the translation of type E does not include types no type error will be reported. Behavioral retains typed methods but adds behavioral casts on untyped methods. The concrete semantics retains typed methods, and adds a subtype cast when a variable of type $\star$ is passed to a method that expects and E.

Source:

class E { m(x: D): D {x} }

Optional:

class E { m(x: $\star$): $\star$ {x} }

Transient:

class E { m(x: $\star$): $\star$ {$\langle$E$\rangle$ x; $\langle\star\rangle$ x} }

Behavioral:

class E { m(x: E): E {x} }

Concrete:

class E { m(x: E): E {x} }

Figure 13 Translation of E in litmus test 3.

Fig. 13 presents the translation of class E. For the transient semantics, when x is cast to E, all of the types on E are erased. Casting to E is tantamount to asking for the existence of the method m. In contrast, the concrete semantics retains the types of m. A cast to E is equivalent to checking if a method m that takes and returns an E exists. This comes at the cost of the ability to migrate between untyped and typed code. Suppose that both the optional and concrete versions of E existed, under a different name F. In that program, only the concrete version of E could be used with the concrete version of F. Despite implementing the same behavior, Behavioral uses the same representation for E as concrete. The behavioral cast allows to use any value that behaves like an E.

To examine the operation of the behavioral cast in more detail, figure 14 depicts the wrapper classes generated at the cast from C to $\star$ and from it to E. Class C_1 takes an instance of C and makes it safe against use as $\star$. In behavioral, no typed invocations can be made on a value that was cast to $\star$ (and not cast to some type somewhere); only untyped invocations are allowed. As a result, the wrapper need only generate an untyped version of C's method a, which calls the underlying C instance's a (adding suitable casts). The second wrapper class C_2 takes the C_1 wrapper and casts it back to E. This wrapper takes the untyped implementation of a and wraps it again, calling it with an argument cast to $\star$ and casting the return to D.

5.4 Discussion

These translations make explicit the semantics of the various approaches. The programs can get stuck at dynamic invocation and casts. Inspecting where these are inserted in the various translations gives a precise account of what constitutes an error in each gradual type system.

$\textbf{class}\ \mathsf{C}\,\{\ \mathsf{a}(\mathsf{x}\colon \mathsf{C})\colon \mathsf{C}\ \{\mathsf{x}\}\ \}$ $\qquad$ $\textbf{class}\ \mathsf{E}\,\{\ \mathsf{a}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{D}\ \{\mathsf{x}\}\ \}$

$$
\begin{array}{l}
\textbf{class}\ \mathsf{C}_1\,\{\,\mathsf{that} : \mathsf{C} \\
\qquad\qquad \mathsf{a}(\mathsf{x}\colon \star)\colon \star\ \{\blacktriangleleft \star \blacktriangleright\ \mathsf{this.that.a}_{\mathsf{C}\to\mathsf{C}}(\blacktriangleleft \mathsf{C} \blacktriangleright\ \mathsf{x})\}\,\} \\
\textbf{class}\ \mathsf{C}_2\,\{\,\mathsf{that} : \mathsf{C}_1 \\
\qquad\qquad \mathsf{a}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{D}\ \{\blacktriangleleft \mathsf{D} \blacktriangleright\ \mathsf{this.that.a}_{\star\to\star}(\blacktriangleleft \star \blacktriangleright\ \mathsf{x})\}\,\}
\end{array}
$$

Figure 14 Behavioral wrappers.

Our account of the behavioral approach matches its implementation in Typed Racket. But one could imagine a slightly less restrictive implementation, one which does not have a check for method names at wrapper creation. That check is pragmatic but perhaps too strict – it will rule out programs that may be fine just because a method is missing. One could have a wrapper that simply reports an error if a missing method is called.

Performance is a perennial worry for implementers of static type systems. It is difficult to provide guesses of how a highly optimizing language implementation will perform, as these implementations are likely to optimize away the majority of the casts and the dynamic dispatches. Consider the progress in the performance of Typed Racket reported since the publication of Takakiwa et al. paper [21]. What we can tell by looking at the translations is that in the optional approach there is no obvious benefit or cost to having type annotations. The transient approach has checks on reads, and typically reads are frequent, so, this seems like an unlucky design choice. Furthermore, those checks are needed even if the entire program is typed. Both concrete and behavioral can benefit from type information in typed code. The difference is that cost of boundary crossing are low for concrete as they end up being a tqg check, whereas behavioral requires allocation of wrapper. Wrappers have other costs as they may complicate the task of devirtualization and unboxing.

6 Conclusion

This paper has introduced KafKa, a framework for comparing the design of gradual type systems for object-oriented languages. Our approach is to provide translations from common surface language with different gradual semantics into KafKa. These translations highlight the different run-time enforcement strategies deployed by the languages under study. The differences between gradual type systems are highlighted explicitly by the observable differences of their behavior in our litmus tests, demonstrating how there is no consensus on the meaning of error. These litmus tests motivated the need to have a common framework to explore the design space.

KafKa demonstrates that to express the different gradual approach, one needs a calculus with two casts, one structural and one behavioral, two invocations forms, one dynamic and one static, the ability to extend the class table at run-time, and wrappers that expose their underlining unwrapped methods. We provide a mechanized proof of soundness for KafKa that includes run-time class generation. We also demonstrate that KafKa can be straightforwardly implemented on top of a stock virtual machine.

A open question for gradual type system designers is performance of the resulting implementation. Performance remains a major obstacle to adoption of approaches that attempt to provide strong guarantees. Under the optional approach, types are removed by the translation, as a result, performance will be identical to that of untyped code. The transient approach checks types at uses, the act of adding types to a program introduces more casts and may slow the program down. This is true even in fully typed code. In contrast, the behavioral approach avoids casts in typed code,. The price it pays for soundness, however, is heavyweight wrappers are inserted at typed-untyped boundaries. Lastly, the concrete semantics also provide strong guarantees and has relatively low overheads, but this comes at a cost in expressiveness.

Going forward there are several issues we wish to investigate. We do not envision that supporting nominal subtyping within KafKa will pose problems, it would only take adding a nominal cast and changing the definition of classes. Then nominal and structural could coexist. A more challenging question is how to handle the intricate semantics of Monotonic Reticulated Python. For these we would need a somewhat more powerful cast operation. Rather than building each new cast into the calculus itself, it would be interesting to axiomatize the correctness requirements for a cast and let users define their own cast semantics. The goal would be to have a collection of user defined pluggable casts within a single framework.

Acknowledgments

The author thank the reviewers of ECOOP, POPL, ESOP, and again ECOOP for thoughtful comments that gradually improved this paper. We are grateful to Benjamin Greenman and Matthias Felleisen for their feedback. This work funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412), the NSF (award 1544542) and ONR (award 503353).

References

1 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

2 Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for smalltalk. *Science of Computer Programming*, 96, 2014. doi:10.1016/j.scico.2013.06.006.

3 Spenser Bauman, Carl Friedrich Bolz-Tereick, Jeremy Siek, and Sam Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *Proc. ACM Program. Lang.*, 1(OOPSLA):54:1–54:24, October 2017. URL: http://doi.acm.org/10.1145/3133878, doi:10.1145/3133878.

4 Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.

5 Gavin M. Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming (ECOOP)*, 2010. doi:10.1007/978-3-642-14107-2_5.

6 Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strnisa, Jan Vitek, and Tobias Wrigstad. Thorn: Robust, concurrent, extensible scripting on the JVM. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1639950.1640016.

7 Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004. doi:10.1145/1167473.1167479.

8 Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1993. doi:10.1145/165854.165893.

9 Luca Cardelli. Amber. In *LITP Spring School on Theoretical Computer Science*, pages 21–47. Springer, 1985.

10 Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM: A comparison of reflective and generative techniques from Scala's perspective. In *Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOLPS)*, 2009. doi:10.1145/1565824.1565829.

11 Robert Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/581478.581484.

12 Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3), 2001. doi:10.1145/503502.503505.

13 Timothy Jones and David J. Pearce. A mechanical soundness proof for subtyping over recursive types. In *Proceedings of the 18th Workshop on Formal Techniques for Java-like Programs*, FTfJP'16, pages 1:1–1:6, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2955811.2955812, doi:10.1145/2955811.2955812.

14 Fabian Muehlboeck and Ross Tate. Sound gradual typing is nominally alive and well. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi:10.1145/3133880.

15 Benjamin C. Pierce and David N. Turner. Local type inference. In *Symposium on Principles of Programming Languages (POPL)*, 1998. doi:10.1145/345099.345100.

16 Gregor Richards, Ellen Arteca, and Alexi Turcotte. The vm already knew that: Leveraging compile-time knowledge to optimize gradual typing. *Proc. ACM Program. Lang.*, 1(OOPSLA):55:1–55:27, October 2017. URL: http://doi.acm.org/10.1145/3133879, doi:10.1145/3133879.

17 Gregor Richards, Francesco Zappa Nardelli, and Jan Vitek. Concrete types for TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:10.4230/LIPIcs.ECOOP.2015.76.

18 Jeremy Siek. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006. http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf.

19 Jeremy Siek and Walid Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, 2007. doi:10.1007/978-3-540-73589-2_2.

20 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined Criteria for Gradual Typing. In Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett, editors, *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5031, doi:10.4230/LIPIcs.SNAPL.2015.274.

21 Asumu Takikawa, Daniel Feltey, Ben Greenman, Max New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *Symposium on Principles of Programming Languages (POPL)*, 2016. doi:10.1145/2837614.2837630.

22 Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2012. doi:10.1145/2398857.2384674.

23 The Dart Team. Dart programming language specification, 2016. http://dartlang.org.

24 The Facebook Hack Team. Hack, 2016. http://hacklang.org.

25 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Symposium on Dynamic languages (DLS)*, 2006. `doi:10.1145/1176617.1176755`.

26 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed Scheme. In *Symposium on Principles of Programming Languages (POPL)*, 2008. `doi:10.1145/1328438.1328486`.

27 Michael Vitousek, Andrew Kent, Jeremy Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Symposium on Dynamic languages (DLS)*, 2014. `doi:10.1145/2661088.2661101`.

28 Michael M. Vitousek, Cameron Swords, and Jeremy G. Siek. Big types in little runtime: Open-world soundness and collaborative blame for gradual type systems. 2017. `doi:10.1145/3009837.3009849`.

29 Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integrating typed and untyped code in a scripting language. In *Symposium on Principles of Programming Languages (POPL)*, 2010. `doi:10.1145/1706299.1706343`.