KafKa: A Common Framework for Gradual Typing of Objects

Benjamin Chung[1], Paley Li[2], and Jan Vitek[3]

1,3 Northeastern University
2 CVUT Prague

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, others can only be caught as the program executes. After a decade of research it is clear that the combination of mutable state, aliasing and subtyping presents challenges to gradual type systems. We introduce KafKa, a class-based object calculus with a static type system, dynamic method dispatch, transparent wrappers and dynamic class generation. KafKa was designed as a common framework for expressing the semantics of various gradually typed object-oriented languages. We demonstrate the usefulness by translating idealizations of four different gradually typed language languages into the calculus and discuss the implications of their respective designs.

1998 ACM Subject Classification Dummy classification – please refer to `http://www.acm.org/about/class/ccs98-html`

Keywords and phrases Dummy keyword – please provide 1–5 keywords

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.-1

1 Introduction

A decade ago, Siek and Taha [11] presented a gradual type system for a variant of Abadi and Cardelli's object-based calculus [1]. Their system featured a dynamic type, denoted $\star$, and a subtype relation that combined structural subtyping with a consistency relation between terms that only differ in dynamic type annotations. Soundness at the boundaries between typed and untyped code is ensured by inserting casts following in their earlier work for functional languages [10].

Despite a decade of hard work, the practical realization of Siek and Taha's elegant idea remains elusive, as their original presentation elided mutable state, one of the most common features of object-oriented programming languages, and adding mutable state has proven to be complex. In conjunction with aliasing and subtyping, mutable state introduces many more points where typed and untyped code can interact, and a wide range of solutions have been put forward to protect these interactions, each of which has different semantics and performance implications.

Four major approaches have arisen to deal with this problem:

- *Optional* type systems, used in languages including Dart [14] and Hack [15], are widely used, but offer little in the way of runtime checking for types.
- *Behavioral* type systems, used by languages like Typed Racket [13, 16] and versions of Reticulated Python [18], are the direct descendants of Siek and Taha's work, and claim good semantics, but have reported poor performance results [12].

- *Transient* type systems, put forward as part of Reticulated Python citesiek14, have been touted to provide better performance than behavioural type systems, but have a different enforcement semantics.
- *Concrete* type systems, part of Thorn [5] and Nom CITEME, provide excellent performance, but suffer from reduced interoperability with untyped code.

The detail lurking behind the curtain of these different languages is that, to attain their benefits and semantic features, each of these approaches has a fundamentally different concept of soundness, with correspondingly different guarantees, correctness properties, and errors. Moreover, the obscuring curtain is a particularly opaque one, as these key details are hidden inside a sea of language-specific detail, making direct comparison impossible.

This paper introduces a core calculus for evaluating gradual type systems for object-oriented languages, called KafKa, designed to illustrate these differences. KafKa is intended to be illustrative of realistic gradually typed language implementations and to clearly demonstrate differences between source language terms translated according to the different gradual type systems. To achieve these goals, we designed KafKa as a statically typed object-oriented language similar to Featherweight Java, augmented with dynamic types and invocations, as well as two kinds of cast, making it easy to see from the static KafKa types what soundness guarantee survives the gradual type translation, while maintaining similarity to gradually-typed languages implemented through translation to Java Bytecode, among others.

Using a common source language, we present a translation for each of the four gradual type systems into KafKa, making the guarantees provided by the gradual type system explicit through the traditionally-sound KafKa type system, while illustrating the cast insertion strategy needed to provide that guarantee in the translation itself, and providing an evaluation semantics for the gradually type system via the KafKa dynamic semantics.

We provide a soundness proof for KafKa, mechanized in Coq, and a proof-of- concept implementation of KafKa on top of the .NET Common Language Runtime, demonstrating that KafKa provides strong type-based guarantees - and validating discussion of the type guarantees provided by the gradual typing translation - and showing that KafKa can be a guide to implementation, respectively.

2 Background

The intellectual lineage of gradual types can be traced back to attempts to add types to Smalltalk and LISP. A highlight on the Smalltalk side is the Strongtalk optional type system [7], which led to Bracha's notion of pluggable types [6]. For him, types exist solely to catch errors at compile-time, never affecting the runtime behavior of programs. The rationale is that types are an add-on that can be turned off without affecting semantics. In the term of Richards *et al.* [9], an optional type system is *trace preserving*, a if when term e reduces to value a, adding type annotations will never cause e to get stuck. This property is valuable to developers as type annotations will not introduce errors (or affect performance). Optional type systems in wide use include Hack [15], TypeScript [3] and Dart [14].

On the functional side, the ancestry is dominated by the work of Felleisen and his students. The Typed Scheme [17] design that later became Typed Racket is influenced by the authors' earlier work on higher-order contracts [8]. 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, thus the value held in a variable annotated as a C should always behave as a C. To aid debugging,

any departure from the expected behavior of an object, as defined by its behavioral type, must be 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.

Siek and Taha coined the term gradual typing in [10] 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." Their contribution was a formalization of the idea in a lambda calculus with references and a proof of soundness. They defined the type consistency relation $\mathsf{t} \sim \mathsf{t}'$ which states that types that agree on non-$\star$ positions are compatible. In [11] the authors extended their result to a stateless object calculus and combined consistency with structural subtyping, but extending this approach to mutable objects proved challenging. Reticulated Python [18] is a compromise between soundness and efficiency. The language has three modes: the *guarded* mode behaves as Racket with contracts applied to values. The *transient* mode performs shallow checks on reads and returns, only validating if the value obtained has matching method names. The *monotonic* mode is fundamentally different. 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.[1]

Other noteworthy systems include Gradualtalk [2], C# 4.0 [4], Thorn [5], and StrongScript [9]. 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, in practice, rarely used). C# 4.0 adds the type `dynamic` to C# and dynamically resolved method invocation. C# has thus 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.

Fig. 1 reviews gradual type systems with publicly available implementations. All languages here are class-based, except TypeScript which has both classes and plain JavaScript objects. That choice does not appear to be crucial to the approach, but since type declarations are needed, classes are an obvious way to introduce them. Most languages base subtyping on explicit name-based subtype declarations (nominal subtyping, rather than on structural similarities. TypeScript uses structural subtyping, but does not implement a runtime check for it. Anecdotal evidence suggests that structural subtyping is rarely needed in TypeScript [9], and in fact, StrongScript extends TypeScript changing subtyping back to nominal for improved performance. While nominal subtyping leads to more efficient type casts, the consistency relation used in Reticulated Python is fundamentally structural, it would be nonsensical to use it in a nominal system.[2] For Racket, the heavy use of first-class

[1] A previous version of this paper provided a translation for monotonic. We removed it for space reason and because the authors of Reticulated Python seem to have abandoned the approach.

[2] Consistency relates classes that differ in the number of occurrences of $\star$ in their type signature and not whether they are declared to extend one another. There is no clear way to have consistency and nominal subtyping in the same context.

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

Figure 1 Overview of implemented gradual type systems. (1) Gradualtalk has optional structural constraints. (2) Concretely typed expressions are sound in C#, Thorn and StrongScript.

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

Optional types are the default execution mode for Dart, Hack and TypeScript. Transient Reticulated Python is, in some senses, optionally typed as any value can flow into a variable regardless of its type annotation, leading to its "open world" soundness guarantee [18]. In Thorn and C#, primitives are concretely typed; they can be unboxed without tagging. The choice of casts follows from other design decisions. Languages with concrete types naturally tend to use subtype casts 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. A fitting analogy are the stack traces printed by Java when a program terminates abruptly. Developers working in, e.g, C++ must run their program in a debugger to obtain the same information. Stack traces have little runtime cost because they piggyback on precise exceptions. Recording blame has a cost, but no data exists on its performance impact. We do not consider blame further in this paper.

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 tend to suffer prohibitive slow downs in pathological cases. Compiler optimizations for reducing these overheads are an active research topic.

3 KafKa: A Core Calculus

KafKa is a class-based, statically typed, object-oriented calculus. The distinctive features of the calculus are its support for both typed and untyped method invocation, as well as dynamic class generation through explicit class tables. The features of KafKa are modeled on common compilation targets for object-oriented language such as the JVM's Java Bytecode or the .NET CLR's Common Intermediate Language, both of which are typed intermediate languages with support for untyped method calls (through reflection) and class generation (via dynamic loading). Our design is guided by the intuition that the semantics of gradually-typed object-oriented languages can be represented by translation to a common representation through the introduction of *casts* and *wrappers*.

$$
\begin{array}{llllll}
\mathsf{e} & ::= & \mathsf{x} \mid \mathsf{this} \mid \mathsf{that} & \mid \mathsf{this.f} & \mathsf{k} & ::= \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\} \\
 & & \mid \mathsf{this.f} = \mathsf{e} & \mid \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{e}) & \mathsf{md} & ::= \quad \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\} \\
 & & \mid <\mathsf{t}>\mathsf{e} & \mid \blacktriangleleft\,\mathsf{t}\,\blacktriangleright\,\mathsf{e} & \mathsf{fd} & ::= \quad \mathsf{f}\colon \mathsf{t} \\
 & & \mid \textbf{new}\ \mathsf{C}(\mathsf{e}_1..) & \mid \mathsf{e@m}_{\star\to\star}(\mathsf{e}) & \mathsf{t} & ::= \quad \star \mid \mathsf{C}
\end{array}
$$

Figure 2 KafKa Syntax.

3.0.0.1 Syntax

KafKa's syntax is given in Fig. 2. Types consist of class names C, D and the dynamic type, written $\star$. Class definitions have a class name and sequences of field and method definitions, $\textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}$. Field definitions consist of a field and its type, $\mathsf{f}\colon \mathsf{t}$. Method definitions have (for simplicity) a single argument and an expression, denoted $\mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\}$. KafKa supports a limited form of overloading, allowing both a typed implementation and an untyped implementation for each method. Fields are private to objects, and can be accessed only from the object's scope; reading a field is denoted this.f and writing a field is denoted $\mathsf{this.f} = \mathsf{e}$. A statically resolved call, denoted $\mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$, is guaranteed to succeed as the type system ensures that the expression e will evaluate to an object of a class which has a method $\mathsf{m}(\mathsf{t})\colon \mathsf{t}'$. A dynamically resolved call, $\mathsf{e@m}_{\star\to\star}(\mathsf{e}')$, can get stuck if the object that e evaluates to does not have a method $\mathsf{m}(\star)\colon \star$. We let meta-variables x ranges over variable names, m and f range over methods and fields respectively; this is a distinguished identifier representing a method receiver, while that is a distinguished field name that will be used in wrapper classes. Providing two different cast mechanisms is a key feature of the calculus. The former, the *structural cast*, $<\mathsf{t}>\ \mathsf{e}$, denotes the usual subtype cast that dynamically type-checks its argument. The latter, the *behavioral cast*, $\blacktriangleleft\,\mathsf{t}\,\blacktriangleright\,\mathsf{e}$, rather than type-checking the argument at runtime, builds a wrapper around it. The wrapper then ensures that all the successive requests to the object will be understood (or raise an error). The design of the behavioral cast is intricate, and deserves its own section below. State is represented via a heap σ mapping addresses ranged over by a to objects denoted $\mathsf{C}\{\mathsf{a}_1..\}$. In some cases we allow the body of a method to be multiple expressions separated by a semicolon, e;e, this is syntactic sugar (it can be encoded by creating a helper object with two fields).

3.0.0.2 Static semantics

A well-formed program, denoted $\mathsf{e}\ \mathsf{K}\ \checkmark$, consists of an expression e and a class table K, where each class k is well-formed and e is well-typed with respect to K. A class is well-formed if all

its fields and methods are well-typed and it has at most two definitions for any method m, one typed $\mathsf{m}(\mathsf{x}\colon \mathsf{C})\colon \mathsf{D}\ \{\mathsf{e}\}$ and one untyped $\mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{e}\}$. The static semantics of KafKa is mostly standard; the complete set of rules is in Appendix. The subtype relation, $\mathsf{M}\ \mathsf{K} \vdash \mathsf{t} <\colon \mathsf{t}'$, shown in Fig. 3, allows for recursive structural subtyping, using the environment M as in the amber rule. The dynamic type $\star$ is a singleton in the subtype relation, it is neither a super or a sub-type of any other type. The notation $\mathsf{md} \in \mathsf{K}(\mathsf{C})$ denotes the method definition md occurring in class C and class table K. Fields are hidden from the class type signature, so subtyping is limited to method definitions. Key type rules are in Fig. 4. Method calls use syntactic disambiguation to select between methods, based on type signature, where dynamic method invocation (@) always invokes methods of type$\star$. A dynamically resolved call places no requirements on the receiver or argument, and returns a value of type $\star$. A statically resolved call has the usual type requirements on arguments. The two subtype cast rules are similar; they ensure the value has the cast-to type. However, their enforcement mechanisms are very different, as will be discussed later.

3.0.0.3 Dynamic Semantics

The small step operational semantics for KafKa appears in Fig. 5. To resolve the this reference in field accesses, the syntax of expressions is extended at runtime with forms

$$\mathsf{e} ::= \ldots \mid \mathsf{a} \mid \mathsf{a.f} \mid \mathsf{a.f} = \mathsf{e}$$

where a is a reference. The static typing of references (Fig. 4) can be either the class of the object they map to in the heap σ or the dynamic type $\star$. We also use evaluation contexts, E, defined as follows

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

The dynamic semantics is defined over *configurations*: triples $\mathsf{K}\ \mathsf{e}\ \sigma$, where K is a class table, e is an expression and σ is a heap. A configuration evaluates in one step to a new configuration, $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$; the new configuration may include a new class table built by extending the previous table with new classes. Calling forms specify the typing of the method to resolve overloading; this is always $\star \to \star$ for dynamic calls, but can be either $\mathsf{C} \to \mathsf{D}$ or $\star \to \star$ for static calls. Structural casts to $\star$ always succeed while structural casts to C check that the runtime object is an instance of a subtype of C. Evaluation contexts are deterministic and enforce a strict evaluation order.

A behavioral cast $\blacktriangleleft\ \mathsf{t}\ \blacktriangleright\ \mathsf{a}$ creates a wrapped object a' which dynamically checks that object a behaves as if it was of type t. We refer to the type of a as the *source* type, and to

$$\frac{\mathsf{C} <\colon \mathsf{D} \in \mathsf{M}}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{C} <\colon \mathsf{D}} \qquad \frac{\begin{array}{c}\mathsf{M}' = \mathsf{M}\ \ \mathsf{C} <\colon \mathsf{D} \\ \mathsf{md} \in \mathsf{K}(\mathsf{D}) \implies \mathsf{md}' \in \mathsf{K}(\mathsf{C})\ .\ \mathsf{M}'\ \mathsf{K} \vdash \mathsf{md} <\colon \mathsf{md}'\end{array}}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{C} <\colon \mathsf{D}}$$

$$\frac{\begin{array}{c}\mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_1' <\colon \mathsf{t}_1 \\ \mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_2 <\colon \mathsf{t}_2'\end{array}}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1)\colon \mathsf{t}_2\ \{\mathsf{e}\} <\colon \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1')\colon \mathsf{t}_2'\ \{\mathsf{e}'\}}$$

Figure 3 KafKa subtyping.

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

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

Figure 4 KafKa static semantics (excerpt).

K	**new** $\mathsf{C}(\mathsf{a}_1..)$	σ	$\to$	K	a'	σ'	**where**	a' fresh $\quad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\mathsf{a}_1..\}]$
K	$\mathsf{a}.\mathsf{f}_i$	σ	$\to$	K	a_i	σ	**where**	$\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}_i, \mathsf{a}_n \ldots\}$
K	$\mathsf{a}.\mathsf{f}_i = \mathsf{a}'$	σ	$\to$	K	a'	σ'	**where**	$\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}_i, \mathsf{a}_n \ldots\}$
								$\sigma' = \sigma[\mathsf{a} \mapsto \mathsf{C}\{\mathsf{a}_1, \ldots \mathsf{a}', \mathsf{a}_n \ldots\}]$
K	$\mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{a}')$	σ	$\to$	K	e'	σ	**where**	$\mathsf{e}' = [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}$
								$\mathsf{m}(\mathsf{x}: \mathsf{t}_1) : \mathsf{t}_2\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C})$
								$\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\} \quad \emptyset\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}_1$
								$\emptyset\ \mathsf{K} \vdash \mathsf{t}_2 <: \mathsf{t}'$
K	$\mathsf{a}@\mathsf{m}_{\star\to\star}(\mathsf{a}')$	σ	$\to$	K	e'	σ	**where**	$\mathsf{e}' = [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}$
								$\mathsf{m}(\mathsf{x}: \star) : \star\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C})$
								$\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\}$
K	$<\star>\mathsf{a}$	σ	$\to$	K	a	σ		
K	$<\mathsf{D}>\mathsf{a}$	σ	$\to$	K	a	σ	**where**	$\emptyset\ \mathsf{K} \vdash \mathsf{C} <: \mathsf{D} \quad \sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\}$
K	$\blacktriangleleft\ \mathsf{t}\ \blacktriangleright\ \mathsf{a}$	σ	$\to$	K′	a'	σ'	**where**	$\mathsf{K}'\ \mathsf{a}'\ \sigma' = \mathsf{bcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K})$
K	$\mathsf{E}[\mathsf{e}]$	σ	$\to$	K′	$\mathsf{E}[\mathsf{e}']$	σ'	**where**	$\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$

Figure 5 KafKa dynamic semantics.

the type t as the *target* type. This cast is generative as it creates both a new class (for the wrapper object) and a new instance of that class (the wrapper itself). Its dynamic semantics is reported in Fig. 6 and Fig. 7. When the target type is a class, say C′, the bcast function wraps a reference a with an instance of a freshly generated wrapper class D that abides by the interface of C′ or gets stuck. The function allocates the wrapper in the heap and updates the class table. The cast performs a check that the source type has at least all the method names requested by the target type. Also the cast is not defined for classes with overloaded methods (nodups prevents it from being used in this case). This prevents ambiguity in method resolution for wrapped objects. When the target type is ⋆, the cast allows a to act like an instance of ⋆.

Wrappers play two roles: they check that object being wrapped has the method expected by the target type, and they are adapters that ensure KafKa code is well-typed. Consider for example a ◀ D ▶ **new** (C) where C and D are defined as follows.

```
class C {                    class D {
    m(x: ⋆): ⋆ {x}               m(x: D): D {x}
}                            }
```

The cast will check that the instance of C has the method expected by D, in this case this is the sole method m. Importantly, it does not check whether argument and return types match (in this example, they do not). The second role of the cast is for soundness of KafKa,

the value returned by the cast must be a subtype of the target type D. This is achieved by generating a new class with the right method signatures.

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

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

Figure 6 Behavioral casts.

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

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

Figure 7 Wrapper class generation.

Wrapper class generation is implemented by the W and $\mathsf{W}\star$ functions. The grey background is used to identify generated code. Their first three arguments C, $\mathsf{md}_1..$, $\mathsf{md}'_1..$ are the source type and its methods definitions, and the method definitions of the target type. The fourth argument, D, is the class name of the wrapper class being built; this is always a fresh name. The last argument is the that field name, which is also fresh. The function builds a new wrapper class D. The field that stores a reference to the target object and has thus type C. The invocation of a method that appears in $\mathsf{md}'_1..$ is forwarded to the corresponding method invocation in md_1, except that the arguments are protected by behavioral casts following the interface in $\mathsf{md}_1..$ and the return type following the interface in $\mathsf{md}'_1...$ Methods defined in the source type but missing from $\mathsf{md}'_1..$ are added to the wrapper class and simply redirected to corresponding method in the wrapped object. Wrapping an object so that it behaves as $\star$ is simpler, as the wrapper systematically protects the input type and casts back the return type to $\star$. Only methods existing in the source type are wrapped. The behavioral cast reduction rule satisfies a property instrumental to our design: a wrapper class generated for a target type D, is always a subtype of D. This will allow us to refer to both unwrapped and wrapped objects via the original, source language, types. The wrapper generation function will always produce a well-formed class for a target type D. The reason being that the wrapper class only contain methods from either the source or the target type. The wrapper class will also never contain any duplicates as it can only contain methods that exists in the source type.

Furthermore, at most the wrapper class will contain every method of the source type.

3.1 Type soundness

We have mechanized the type soundness proof for KafKa in Coq[3]. In Coq syntax, the conditions for KafKa's type soundness theorem is stated as follows:

```
∀ (k:ct) (s:heap) (e:expr) (t:type),
  (*condition 1*) WellFormedState k e s →
  (*condition 2*) HasType nil s k e t →
```

Our Coq mechanization is equivalent to our formalism. In particular, the inference rules $\Gamma\ \sigma\ \mathsf{k} \vdash \mathsf{e} : \mathsf{t}$ and $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ is equivalent to the Coq functions *HasType* and *WellFormedState* respectively. Note that in each of the three error stuck states, cases 3 through 5, the actual stuck expression can be inside some evaluation context `E`. The *equivExpr* function fills in the hole in `E` with the given expression.

The theorem states that for any class table (`k`), heap (`s`), expression (`e`), and type (`t`), if the state is well formed (condition 1), and if the top-level expression `e` has type `t` (condition 2), then the program can only make one of the following five choices:

```
  (*case 1*) (∃ a:ref, e = Ref a) ∨
```

Case 1 The program is already fully evaluated. Since KafKa has only one value, reference (ref in our Coq formalization), we can say that there exists some reference that is equal to our entire program, or `∃ a : ref, e = Ref a`.

```
  (*case 2*) (∃ (e′:expr) (s′:heap)(k′:ct),
        Steps k e s k′ e′ s′ ∧
        WellFormedState k′ e′ s′ ∧
        HasType nil s′ k′ e′ t ∧
        retains_references s s' ∧
        ct_ext k k') ∨
```

Case 2 The program can take a step. In this case, the program will step to some new configuration (expression `e′`, heap `s′`, and class table `k′`) that is both itself well formed and that retains all of the entries and types in the old heap `s` (expressed using *retains_references*) and class table `k` (expressed using *ct_ext*).

```
  (∃ E:EvalCtx,
  (*case 3*) (∃ (a:ref) (m:id) (a′:ref) (C:id) (aps:list ref),
        (e = equivExpr (DynCall (Ref a) m (Ref a′)) E) ∧
        In (a, HCell(C, aps)) s ∧
        ∀ x e, ~ (In (Method m x Any Any e) (methods C k))) ∨
```

Case 3 The program could be stuck at a dynamic call expression. The dynamic call expression of the form *DynCall* (*Ref* `a`) `m` (*Ref* `a′`) can get stuck if `a` refers to an instance of class `C` in `s` that does not contain a method `m` to call.

[3] https://github.com/BenChung/GradualComparison/tree/master/impl

```
(*case 4*) (∃ (t′:type) (a:ref) (C:id) (aps:list ref),
      (e = equivExpr (SubCast t′ (Ref a)) E) ∧
      In (a, HCell(C,aps)) s ∧
      ((Subtype empty_mu k (class C) t′) -> False)) ∨
```

Case 4 The program could have gotten stuck on a subtype cast. In an expression of the form *SubCast* $\mathtt{t}'$ (*Ref* a), the program can get stuck if s maps a to an object of class C, and if C is not a subtype of $\mathtt{t}'$.

```
(*case 5*) (∃ a aps C C′,
      e = equivExpr (BehCast (class C') (Ref a)) E ∧
      In (a, HCell(C, aps)) s ∧
      ~(incl (method_names C′ k) (method_names C k))))
```

Case 5 The program may get stuck on a behavioral cast, of the form *BehCast* (*class* $\mathtt{C}'$) (*Ref* a), if C, the class of a in s does not have all of the methods needed to implement $\mathtt{C}'$.

We have proven that a KafKa program will not get stuck, except if it encounters a dynamic call to an object that does not have the required method, tries to use subtype cast on a value that is not actually a subtype, or attempts to force an object to act like another object, however with missing required method signature. The KafKa type system provides a guarantee of progress, most expressions are guaranteed to step. This localizes where KafKa programs could potentially encounter runtime errors, which helps reasoning about where gradually typed programs go wrong.

3.2 Implementation

We designed KafKa to have a close correspondence to the intermediate languages of commercial VMs. To validate this aspect of our design, we implemented a small compiler from KafKa to C# and the CLR, alongside a runtime that provides the behavioral cast operation.[4].

The only substantial challenge in the development relates to one of our earliest design choices: structural subtyping. Structural typing is key for the consistent subtyping used in Reticulated Python, but few virtual machines have built-in support for it. Our implementation converts structural types to nominal types, maintaining semantic equivalence. We do this by whole program analysis. Consider a class table K, and assume that $\mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$ for some t and t'. C# already handles the case where t or t' is $\star$, via its dynamic type, which has the same semantics as $\star$ in KafKa. If $\mathsf{t} = \mathsf{C}$ and $\mathsf{t}' = \mathsf{D}$, then we generate the interface ID, which the translated version of C implements. ID has the same methods as D, allowing the same operations to be performed on its instances. We then refer to D by ID in the generated C# type signatures and casts, which then allows our translation of C to be used wherever a D is expected, satisfying subtyping. If we apply this to every pair of types C and D where the subtyping relation holds, the C# subtyping relation ends up mirroring the KafKa one exactly. C# has another quirk in its type system, namely it does not allow for covariance or contravariance in interface implementation, whereas KafKa's subtyping allows both. However, C# does allow for *explicit implementations*, which we use to implement covariant and contravariant interface implementation, by explicitly calling out the interface method to implement and forwarding to the actual implementation.

[4] https://github.com/BenChung/GradualComparison/tree/master/netImpl/kafkaimpl

The goal of this implementation was to validate the choice of features included in KafKa by showing that they match those provided in modern VMs. For most of KafKa's functionality, the implementation was trivial.

4 Litmus test

One of the salient implications of the various approaches to gradual typing is that they have different notions of what is an erroneous program. This section proposes a litmus test that can distinguish the four type systems. The three programs of Fig. 8 are statically well-typed in all four approaches to gradual type systems.[5] Furthermore, in an untyped language all three programs would execute without error. Depending on the runtime enforcement strategy, some of these programs will halt with a runtime error due to inserted casts. Different type systems will report a different number of errors, and will stop at different points. Fig. 8 presents three programs, written in source language, which can be translated to KafKa following the four different approaches that will be presented in the next section. The programs consist of a class table and a top-level expression.

	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 8 Gradual typing litmus test.

Optional. An optional type system, which simply erases all of the type annotations at runtime, will run all without error.

Concrete. The guarantee provided by the concrete semantics is that only statically checked values can be of a type, implying that untyped values can never be passed in placed of a typed one. This strong requirement causes the concrete semantics to reject all three programs. **L1** fails as it attempts to pass an A instance in place of an instance of I, which are incompatible. **L2** also fails despite A and I's method now being of the same name, the types are statically incompatible due to Q not being a static super type of A. **L3** fails as it tries to pass a C as an E, in the call to m inside of N in F, impossible under the concrete semantics as C is not a subtype of E.

[5] For completeness we implemented them in TypeScript, Typed Racket, Thorn and Reticulated Python, code is in appendix.

Behavioral. In contrast to the concrete semantics, the behavioral semantics does allow untyped values to be passed under a type if they are structurally compatible, but dynamically ensures that the untyped values to respect the type. Like the concrete semantics, **L1** is rejected as A's method, m, could never be used as I's n. **L2**, however, executes without error, as while A.m does not properly implement I.n, this is never observed dynamically, and the value is therefore observationally of the ascribed type. **L3** is rejected by the behavioral semantics, as when the top-level expression calls a on a C that has been cast to an E, the check that the value (another instance of C) is structurally compatible with the required type (D), fails.

Transient. The transient semantics further weakens the guarantee, retaining the structural checks at casts of the behavioral semantics, but only checks adherence to the locally ascribed type, rather than every potentially applicable type. Transient fails **L1**, for the same reason as the other two type systems, and passes **L2**, for the same reason as the behavioral semantics, as a result, but differs in **L3**. **L3** failed under the behavioral semantics because the return value of n carried the type applied by m with it, forcing the C instance to act like an E. However, the transient semantics does not check this, letting the invocation to a succeed.

5 Translating Gradual Type Systems

We are now ready to translate the four gradual type semantics into KafKa. For each semantics, compilation into KafKa is realized by a translation function that maps well-typed source programs into well-typed KafKa terms, respecting a uniform mapping of source types to KafKa types. The compilation makes explicit which type casts (and, in turn, *dynamic type-checks*) are implicitly inserted and performed by the runtime of each semantics, highlighting the similarities and differences between them. Our source languages, those being translated to KafKa, have no explicit casts, instead silently coercing types at typed-untyped boundaries, allowing code like the following:

```
K   new C().m(new D())      where  K  =  class C {
                                          m(x: ⋆): C { x }
                                        }
                                        class D { }
```

The method m takes an argument of type $\star$ and returns the same value under type C. Without a cast, this operation is patently unsafe, as if m is passed a D, the return type of m will be wrong. However, our source language allows this, instead deferring to the translation to provide a safety guarantee. To avoid unnecessary clutter, we present a common syntax, reported in Fig. 9, for the source language. This defines a simple object calculus similar to KafKa, but without method overloading and, most importantly, cast operations. Lastly, we give a source-level type system for each semantics (notated $\vdash_s$), which are largely identical. To simplify the presentation, we will elide rules identical to those in KafKa, and present only the altered rules.

The type system for the surface languages is straightforward, allowing implicit mixing of static and dynamically typed terms, and is shared between all of the gradual type systems, shown in figure 10. The difference between a gradually typed language and a language with the $\star$ type is the ability to implicitly down-cast from $\star$ to some known type t. The mechanisms which make this operation safe are what differentiate each gradual typing semantics. The naive typing rules would not allow an implicit downcast from $\star$ to a non-$\star$ type, as it is not

$$
\begin{array}{lll}
e ::= x \mid \mathsf{this} & \mid \mathsf{this.f} & k ::= \textbf{class}\ C\,\{\,fd_1..\ md_1..\,\} \\
\mid \mathsf{this.f} = e & \mid e.m(e) & t ::= \star \mid C \\
\mid \mathsf{that} & \mid \textbf{new}\ C(e_1..) & md ::= m(x\colon t)\colon t\ \{e\} \\
 & & fd ::= f\colon t
\end{array}
$$

Figure 9 Common source language.

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

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

Figure 10 Source language type system.

safe. However, to make our surface language a gradually typed one, we need to represent the ability to make some kinds of unsafe type conversion within our type system.

The $K \vdash_s t \Rightarrow t'$ operator is used to denote that type t is convertible to one of type t', and is used both for traditional down-casting and for conversions of $\star$ to non-$\star$ types. Our definition of the type convertibility operator is shown in figure 11, denoting the surface level typing judgment as $\Gamma\,K \vdash_s e : t$, for a term t having type t.

$$
\frac{\cdot\ K \vdash_s t <: t'}{K \vdash_s t \Rightarrow t'}
\qquad
\frac{}{K \vdash_s t \Rightarrow \star}
\qquad
\frac{}{K \vdash_s \star \Rightarrow t}
$$

Figure 11 Optional type convertibility.

In the source language, it is possible to pass subtypes as arguments and new field values (STGC-SUB), as in the KafKa type system. However, the source language compatibility relation adds two new cases, STGC-TOANY and STGC-ANYCONC, allowing typed values to be used in place of untyped values and vice versa respectively. If the source program type checks correctly, we can proceed with translation to KafKa. For each of the four semantics, we provide a translation from this common, gradually typed, source language to the statically-typed KafKa target language, illustrating how the guarantees provided by each system influence semantics and performance.

Fig. 12 presents the translation for field access and variables, with notable exceptions in the transient and optional semantics, these operations are translated verbatim. The grey background is used to identify translated KafKa terms. For the optional semantics, we have eliminated the type of the this introduction form by casting it to $\star$. The optional semantics make no other alterations to these terms, as it has no guarantee to ensure. Transient does not guarantee the values passed to functions or field assignment respect the declared type of those expressions. Class translation inserts checks for invalid variables upon method entry, so while

Optional

$$\mathcal{O}[\![\mathsf{x}]\!] = \mathsf{x}$$
$$\mathcal{O}[\![\mathsf{this}]\!] = <\star>\,\mathsf{this}$$
$$\mathcal{O}[\![\mathsf{this.f}]\!] = \mathsf{this.f}$$

Transient

$$\mathcal{T}[\![\mathsf{x}]\!]_\Gamma = <\mathsf{t}>\,\mathsf{x} \quad \textbf{where} \quad \mathsf{K}, \Gamma \vdash \mathsf{x} : \mathsf{t}$$
$$\mathcal{T}[\![\mathsf{this}]\!]_\Gamma = \mathsf{this}$$
$$\mathcal{T}[\![\mathsf{this.f}]\!]_\Gamma = <\mathsf{t}>\,\mathsf{this.f} \quad \textbf{where} \quad \mathsf{K}, \Gamma \vdash \mathsf{this} : \mathsf{C}, \; \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C})$$

Behavioral

$$\mathcal{B}[\![\mathsf{x}]\!]_\Gamma = \mathsf{x}$$
$$\mathcal{B}[\![\mathsf{this}]\!]_\Gamma = \mathsf{this}$$
$$\mathcal{B}[\![\mathsf{this.f}]\!]_\Gamma = \mathsf{this.f}$$

Concrete

$$\mathcal{C}[\![\mathsf{x}]\!]_\Gamma = \mathsf{x}$$
$$\mathcal{C}[\![\mathsf{this}]\!]_\Gamma = \mathsf{this}$$
$$\mathcal{C}[\![\mathsf{this.f}]\!]_\Gamma = \mathsf{this.f}$$

Figure 12 Translations for variables and field access.

the transient translation has to insert casts to appease the KafKa type system, these casts will never fail. However, fields are not protected and requires a guard after dereferencing, as values can be passed through them to violate their types.

Optional

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

Transient

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

Behavioral

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

Concrete

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

Figure 13 Translations for field assignment.

Field assignment, presented in Fig. 13, illustrates the distinct continuum provided by each gradual type system. Under the optional semantics, we simply translate the assignment directly, with no guards or additional checks. However, in the three remaining semantics, the behavior differs depending on the strength of the guarantees. Under the transient semantics, field assignment is unchecked, deferring type checking to dereferencing. As a result, we use the analytic translation operator to translate the argument according to the transient semantics. The type of the field is erased by targeting the $\star$ type in the translation. Under the behavioral and concrete semantics, type consistency is guaranteed for the heap, and the translation for the argument expression is done with the known type of the field. One observation from these translations is that the behavioral and concrete semantics demand the correct type to be returned from the argument expression, while the transient semantics only needs to be well-formed. As a consequence, the behavioral and concrete semantics will raise an error when assigning a value of the wrong type to a field, whereas the transient semantics will rise an error upon dereference.

Fig. 14 presents the translation for the object creation expression. Apart from the optional semantics, the translation for the object creation expression is almost identical between the transient, behavioral, and concrete semantics. The only difference lies in the recursive translation call for the arguments of the object, whereby each semantics call their respective translation. The optional semantics require an additional $\star$ cast easing any typing information from the object.

Optional

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

Transient

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

Behavioral

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

Concrete

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

Figure 14 Translations for object creation.

Transient

$$\mathcal{T}(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = \mathsf{e}' \quad \textbf{where } \mathsf{K}, \Gamma \vdash \mathsf{e} : \mathsf{t}' \ \ \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t} \ \ \mathsf{e}' = \mathcal{T}[\![\mathsf{e}]\!]_\Gamma$$
$$\mathcal{T}(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = <\mathsf{t}> \mathsf{e}' \quad \textbf{where } \mathsf{K}, \Gamma \vdash \mathsf{e} : \mathsf{t}' \ \ \mathsf{K} \vdash \mathsf{t}' \not<: \mathsf{t} \ \ \mathsf{e}' = \mathcal{T}[\![\mathsf{e}]\!]_\Gamma$$

Behavioral

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

Concrete

$$\mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = \mathsf{e}' \quad \textbf{where } \mathsf{K}, \Gamma \vdash \mathsf{e} : \mathsf{t}' \ \ \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t} \ \ \mathsf{e}' = \mathcal{C}[\![\mathsf{e}]\!]_\Gamma$$
$$\mathcal{C}(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = <\mathsf{t}> \mathsf{e}' \quad \textbf{where } \mathsf{K}, \Gamma \vdash \mathsf{e} : \mathsf{t}' \ \ \mathsf{K} \vdash \mathsf{t}' \not<: \mathsf{t} \ \ \mathsf{e}' = \mathcal{C}[\![\mathsf{e}]\!]_\Gamma$$

Figure 15 Translation for type requirement.

Figure 15 depicts the translation for type requirement. For this translation, we are given the expression to translate, the environment to translate against, and the KafKa type of the resulting expression. Under the optional system, all terms have type $\star$, rendering a translation for type requirement redundant. The transient and concrete semantics use standard dynamic subtyping. The concrete semantics use classes defined by the programmer, while the transient semantics alters them during translation. Under the transient semantics, classes are rewritten with dynamic types in place of static types in functions and field. This causes the subtyping cast in the context of the transient class translation to check for the existence of a method of the required name during dynamic subtyping, rather than required to ensure the entire class is type compatible. The behavioral semantics has the same untyped invocation as transient, but provides a stronger guarantee. The behavioral semantics guarantee a typed receiver respects the type it is under, allowing the invocation to call the method under the expected type and not to verify the return type. An ill-typed return would be caught by a wrapper before it reaches the caller.

Figure 16 depicts the translation of function invocation for the four semantics. For the optional semantics, we do not know if the function exists over the object. The translation compensate by having untyped call operator for every function invocation. Each semantics has to make a distinction between safe typed calls and unsafe untyped calls. As a result, a condition is placed over the type of the receiver, using a sound call if it is typed, and an dynamic invocation if not. The transient semantics is almost identically to the optional semantics for function invocation on untyped expressions. A dynamic invocation is generated with a generated argument and $\star$ as the return type, an expression similar to those found in Typescript. In case of sound expressions, the transient semantics will guarantees the existence, but not the types, of methods. In conjunction with class translation, which ensures

Transient

$$\mathcal{T}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}_1'@\mathsf{m}_{\star\to\star}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \star \;\; \mathsf{e}_1' = \mathcal{T}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{T}(\!|\mathsf{e}_2|\!)^\star_\Gamma$$

$$\mathcal{T}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = <\mathsf{D}_2>\mathsf{e}_1'.\mathsf{m}_{\star\to\star}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \mathsf{C} \;\; \mathsf{e}_1' = \mathcal{T}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{T}(\!|\mathsf{e}_2|\!)^\star_\Gamma \;\; \mathsf{m}(\mathsf{D}_1) : \mathsf{D}_2 \in \mathsf{K}(\mathsf{C})$$

Behavioral

$$\mathcal{B}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}_1'@\mathsf{m}_{\star\to\star}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \star \;\; \mathsf{e}_1' = \mathcal{B}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{B}(\!|\mathsf{e}_2|\!)^\star_\Gamma$$

$$\mathcal{B}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}_1'.\mathsf{m}_{\mathsf{D}_1\to\mathsf{D}_2}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \mathsf{C} \;\; \mathsf{e}_1' = \mathcal{B}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{B}(\!|\mathsf{e}_2|\!)^{\mathsf{D}_1}_\Gamma \;\; \mathsf{m}(\mathsf{D}_1) : \mathsf{D}_2 \in \mathsf{K}(\mathsf{C})$$

Concrete

$$\mathcal{C}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}_1'@\mathsf{m}_{\star\to\star}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \star \;\; \mathsf{e}_1' = \mathcal{C}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{C}(\!|\mathsf{e}_2|\!)^\star_\Gamma$$

$$\mathcal{C}[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}_1'.\mathsf{m}_{\mathsf{D}_1\to\mathsf{D}_2}(\mathsf{e}_2') \quad \textbf{where} \quad \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \mathsf{C} \;\; \mathsf{e}_1' = \mathcal{C}[\![\mathsf{e}_1]\!]_\Gamma \;\; \mathsf{e}_2' = \mathcal{C}(\!|\mathsf{e}_2|\!)^{\mathsf{D}_1}_\Gamma \;\; \mathsf{m}(\mathsf{D}_1) : \mathsf{D}_2 \in \mathsf{K}(\mathsf{C})$$

Figure 16 Translations for function invocation.

an untyped method will exist for every typed method, the transient semantics insert a statically-typed call to the $\star$-typed invocation site. However, to be compatible with the dynamic call, the argument still needs to be cast to $\star$ and the return type needs to be checked against the expected type. In the concrete semantics any method can be called statically and dynamically, by generating a dynamic version of all source level methods. Class translation relies on overloading to provide the two versions of each method: one with concrete receiver and the other with optionally typed object. More precisely, given a concrete method md with type $\mathsf{t}_1 \to \mathsf{t}_2$, the concrete translation first generates a method md' with type $\mathsf{kty}(\mathsf{t}_1) \to \mathsf{kty}(\mathsf{t}_2)$ (protecting the the return value with appropriate cast to $\mathsf{kty}(\mathsf{t}_2)$ if necessary). The concrete translation also generates a second method md'' that wraps md' to allow a safe invocation with type $\star \to \star$. Finally, every field $\mathsf{f}{:}\mathsf{t}$ is mapped to the corresponding field $\mathsf{f}{:}\mathsf{kty}(\mathsf{t})$.

Fig. 17 presents the class translation for the four gradual typing semantics, a key part of their functionality. All need to guard against untyped code calling typed receivers through the dynamic invocation, which each handles differently. The optional semantics deals with the problem by eliminating all typed receivers, but the other systems need to check argument and return types. The transient semantics heavily relies upon class translation to achieve the desired semantics, especially with relation to how it handles casts. Under the transient semantics, all types are removed from the method signatures and argument types are instead checked by the method body itself and return types by the caller. Now, with no typed methods, structural subtyping becomes equivalent to checking that all the required method names hold and no more, the cast semantics needed by the transient semantics. The behavioral semantics protects typed methods from untyped callers by means of the generated untyped wrappers, and, as a result, needs no explicit support for such from its translation. The most complex class translation is that of the concrete semantics, which needs to support both typed and untyped callers calling the same class. Here, it uses KafKa's type-based overloading to define a special untyped receiver method, one that is generated if the original method was typed. This receiver method is then responsible for guarding the typed version of the function.

Fig. 18 presents an example of how our translation drives the semantics. We will translate class F's definition using each of the four translations, clearly indicating where the behavior

Optional

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

Transient

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

Behavioral

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

Concrete

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

Figure 17 Translations for class.

of each program. Translation under the optional semantics is extremely simple, all types are erased and all invocations are dynamic. However, the translations become much more complicated when we introduce guarantees. The first, but weakest, gradual type semantics is the transient semantics. Under the transient semantics, all types are erased from the class. Argument and return types are not guaranteed. The transient guarantees only require methods to exist on a type. A statically typed invocation can be made to an untyped method m with only untyped arguments and the return type casted as an $\star$. The weakness of the guarantee provided by the transient semantics means the return type of a method is not guaranteed and the argument types cannot be trusted. The behavioral semantics recover these properties by using the behavioral cast operator. The behavioral semantics inserts wrappers over methods to ensure safety for declared argument and return types. However, there is a cost associated with the behavioral cast, as it is complex and expensive. The concrete semantics is similar to the behavioral semantics in using the typed version of m, and with similarly semantics to the transient semantics. Whereby only the values statically checked are allowed to pass through cast boundary. In Fig. 19 we present the translation of a simply class E. This example aims to illustrate the difference in guarantee provided by the difference semantics.

Source

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

Optional

class F { m(x : ⋆) : ⋆ {x}
n(x : ⋆) : ⋆ {this@$m_{\star\to\star}$(x)} }

Transient

class F { m(x : ⋆) : ⋆ {<E> x; <⋆> x}
n(x : ⋆) : ⋆ {<⋆> x; <⋆> <E> this.$m_{\star\to\star}$(<⋆> <⋆> x)} }

Behavioral

class F { m(x : E) : E {x}
n(x : ⋆) : ⋆ {◀ ⋆ ▶ this.$m_{E\to E}$(◀ E ▶ x)} }

Concrete

class F { m(x : E) : E {x}
n(x : ⋆) : ⋆ {<⋆> this.$m_{E\to E}$(<E> x)} }

Figure 18 Litmus Test 3 Class Translation.

Source

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

Optional

class E { m(x : ⋆) : ⋆ {x} }

Transient

class E { m(x : ⋆) : ⋆ {<E> x; <⋆> x} }

Behavioral

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

Concrete

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

Figure 19 Litmus Test 3 Translation for E.

For the transient semantics, when x is cast to E, all of the types on E are erased by the transient class translation. Casting to E is tantamount to asking for the existence of the method m. In contrast, the concrete semantics retained the types of m. A concrete translation that checks if a class is a subtype of E is equivalent to checking if a method m that takes and returns an E exists. While the concrete semantics provide a stronger type guarantee, it 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 different a name F. In that system, only the concrete version of E could be used with the concrete version of F. Despite implementing the same behavior, the optional version of E would not be able to be used with the concrete version of F, as statically it does not know m would take and return an E. The behavioral semantics is able to use the same representation for E as the concrete semantics. The behavioral cast allows the behavioral semantics to use an optional typed E. In effect, if an optional typed E were passed where a behavioral typed E was expected, then a wrapper, implementing the fully typed E, would be generated on top of the untyped, optional typed E, ensuring type safety.

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 different source language into KafKa. These translations highlight the different runtime enforcement strategies deployed by the languages under study. The differences between gradual type

systems are made explicit with a litmus test that demonstrates observable differences between type systems.

The key features needed in KafKa are two casts, one structural and one behavioral, and the ability to extend the class table at runtime. KafKa was also carefully engineered to support transparent wrappers. We provide a mechanized proof of soundness for KafKa that includes runtime class generation. We also demonstrate that KafKa can be straightforwardly implemented on top of a stock virtual machine.

Going forward there are several issues we wish to investigate further. We do not envision 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 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.

Another open question for gradual type system designers is performance of the resulting implementation. Performance is one of the major obstacles obstructing gradual typing from being incorporated into mainstream languages. To illustrate the impact of performance over the gradual typing semantics, we will discuss an overview of the impact on performance for each gradual typing semantics. Under the optional semantics, types are removed entirely by translation, with the program effectively becoming untyped. As a result, the performance of the optional semantics will be identical to that of a wholly untyped program. The transient semantics checks types at uses (e.g. on function return and entry), the act of adding types to a program introduces more casts and will slow the program down. Additionally, the transient semantics's checks are needed in fully typed code, providing a strict performance detriment without additional optimizations. In contrast, the behavioral semantics does entirely avoid having to insert casts into fully-typed code, because its soundness guarantee extends to variables being of their declared types. The cost it pays for this guarantee, however, is that the declared types are enforced by heavyweight wrappers, inserted at many places throughout the program, a problem noticed by real implementations. Lastly, the concrete semantics is able to achieve a guarantee similar to that of the behavioral semantics, but without the overhead of having to add wrappers, enabled by the strong static guarantee checked for by the cast operator.

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. 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.
4. 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.
5. 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.

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

7 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.

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

9 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.

10 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.

11 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.

12 Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards Practical Gradual Typing. In *European Conference on Object-Oriented Programming (ECOOP)*, 2015. doi:http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.4.

13 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.

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

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

16 Sam Tobin-Hochstadt. *Typed Scheme: From Scripts to Programs*. PhD thesis, 2010.

17 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.

18 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.

A Type system for KafKa

A.1 Well-formedness

The well-formedness judgments for KafKa are defined for programs, classes, methods, fields, and types.

$\boxed{\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark}$ Well-formed program

$$\text{WP}\quad \frac{\emptyset\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash \sigma\ \checkmark \qquad \mathsf{k} \in \mathsf{K} \implies \cdot\mathsf{K} \vdash \mathsf{k}\ \checkmark}{\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark}$$

$\boxed{\sigma\,\mathsf{K} \vdash \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}\ \checkmark}$ Well-formed class

$$\text{WC}\quad \frac{\mathsf{K} \vdash \mathsf{fd}_1..\ \checkmark \qquad \mathsf{this} : \mathsf{C}\ \ \sigma\,\mathsf{K} \vdash \mathsf{md}_1..\ \checkmark \qquad \mathsf{nodups}(\mathsf{md}_1..\ \mathsf{fd}_1..)}{\sigma\,\mathsf{K} \vdash \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}\ \checkmark}$$

$\boxed{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{md}\ \checkmark}$ Well-formed methods

$$\text{WT1}\quad \frac{\Gamma' = \Gamma\ \mathsf{x}\colon\star \qquad \Gamma'\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star \qquad \mathsf{K} \vdash \star\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{m}(\mathsf{x}\colon\star)\colon\star\,\{\mathsf{e}\}\ \checkmark}$$

$$\text{WT2}\quad \frac{\Gamma' = \Gamma\ \mathsf{x}\colon\mathsf{C} \qquad \Gamma'\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C}' \qquad \mathsf{K} \vdash \mathsf{C}\ \checkmark \qquad \mathsf{K} \vdash \mathsf{C}'\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{m}(\mathsf{x}\colon\mathsf{C})\colon\mathsf{C}'\,\{\mathsf{e}\}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash \mathsf{fd}\ \checkmark}$ Well-formed fields

$$\text{WF}\quad \frac{\mathsf{K} \vdash \mathsf{t}\ \checkmark}{\mathsf{K} \vdash \mathsf{f}\colon\mathsf{t}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash \mathsf{t}\ \checkmark}$ Well-formed types

$$\text{WA}\quad \frac{}{\mathsf{K} \vdash \star\ \checkmark} \qquad \text{WC}\quad \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash \mathsf{C}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash \sigma\ \checkmark}$ Well-formed heaps

$$\text{WH}\quad \frac{\mathsf{a}' \mapsto \mathsf{C}\{\mathsf{a}_1..\} \in \sigma \implies \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\} \in \mathsf{K} \ \wedge\ \cdot\sigma\,\mathsf{K} \vdash \mathsf{a}_1 : \mathsf{t}_1\ ..}{\mathsf{K} \vdash \sigma\ \checkmark}$$

A.2 Expression typing

The expression typing judgments for KafKa includes in ascending order as listed in the formalism: variable, untyped address, subsumption, field assignment, field read, static method invocation, dynamic method invocation, object creation, subtype cast, typed address.

$\boxed{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}$ e has type t in environment Γ against heap σ and class table K

$$\text{KT-VAR}\quad \frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{x} : \mathsf{t}} \qquad \text{KT-SUB}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}' \qquad \cdot\,\mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}$$

$$\text{KT-READ}\quad \frac{\Gamma(\mathsf{this}) = \mathsf{C} \qquad \mathsf{f}\colon\mathsf{t} \in \mathsf{K}(\mathsf{C})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{this.f} : \mathsf{t}} \qquad \text{KT-REFREAD}\quad \frac{\sigma(\mathsf{a}) = \mathsf{C}\{..\} \qquad \mathsf{f}\colon\mathsf{t} \in \mathsf{K}(\mathsf{C})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a.f} : \mathsf{t}}$$

$$\text{KT-WRITE}\quad \frac{\Gamma(\mathsf{this}) = \mathsf{C} \qquad \mathsf{f}\colon\mathsf{t} \in \mathsf{K}(\mathsf{C}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{this.f} = \mathsf{e} : \mathsf{t}}$$

$$\text{KT-REFWRITE}\quad \frac{\sigma(\mathsf{a}) = \mathsf{C}\{..\} \qquad \mathsf{f}\colon\mathsf{t} \in \mathsf{K}(\mathsf{C}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a.f} = \mathsf{e} : \mathsf{t}}$$

$$\text{KT-CALL}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C} \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t} \qquad \mathsf{m}(\mathsf{t})\colon\mathsf{t}' \in \mathsf{K}(\mathsf{C})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}') : \mathsf{t}'}$$

$$\text{KT-DYNCALL}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \star}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}@\mathsf{m}_{\star\to\star}(\mathsf{e}') : \star}$$

$$\text{KT-NEW}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}_1 : \mathsf{t}_1.. \qquad \textbf{class}\ \mathsf{C}\,\{\,\mathsf{f}_1 : \mathsf{t}_1..\ \mathsf{md}_1..\,\} \in \mathsf{K}}{\Gamma\,\sigma\,\mathsf{K} \vdash \textbf{new}\ \mathsf{C}(\mathsf{e}_1..) : \mathsf{C}}$$

$$\text{KT-SUBCAST}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash <\mathsf{t}>\,\mathsf{e} : \mathsf{t}} \qquad \text{KT-BEHCAST}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash \blacktriangleleft\,\mathsf{t}\,\blacktriangleright\,\mathsf{e} : \mathsf{t}} \qquad \text{KT-REFTYPE}\quad \frac{\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}'_1..\}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}}$$

$$\text{KT-REFANY}\quad \frac{}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \star}$$

A.3 Dynamic function

The dyn function returns all the methods with $\star$ type for a particular set of signatures of method typing.

$$\text{DYNE}\ \frac{}{\mathsf{dyn}(\cdot) = \cdot} \qquad \text{DYN}\ \frac{\mathsf{dyn}(\mathsf{mt}_1..) = \mathsf{mt}'_1..}{\mathsf{dyn}(\mathsf{m}(\mathsf{t}) : \mathsf{t}\ \ \mathsf{mt}_1..) = \mathsf{m}(\star) : \star\ \ \mathsf{mt}'_1..}$$

A.4 Signature function

The signature function returns method typing signatures (mt) of method definitions (md).

$$\text{SGE}\ \frac{}{\mathsf{signature}(\cdot) = \cdot} \qquad \text{SG}\ \frac{\mathsf{md} = \mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\mathsf{e}\} \qquad \mathsf{signature}(\mathsf{md}_1..) = \mathsf{mt}_1..}{\mathsf{signature}(\mathsf{md}\ \mathsf{md}_1..) = \mathsf{m}(\mathsf{t}) : \mathsf{t}\ \ \ \mathsf{mt}_1..}$$

A.5 Names function

The names function ($\mathsf{names}(\mathsf{fd}_1..)$, $\mathsf{names}(\mathsf{md}_1..)$, $\mathsf{names}(\mathsf{mt}_1..)$) takes either field definitions, method definitions, or method typings, and returns the name of the respective fields or methods.

A.6 Duplicated method names

The nodups function ($\mathsf{nodups}(\mathsf{mt}_1..)$, $\mathsf{nodups}(\mathsf{md}_1..)$) takes either method definitions or method typings, and ensures there are no duplicates.

B Source language well-formedness

Well-formedness for Concrete

The well-formedness judgments for Concrete is similar to the well-formedness judgments of KafKa. The turnstile ($\vdash_s$) of all source language judgment is characterized with s.

$\boxed{\mathsf{e}\ \mathsf{K}\ \checkmark_s}$ Well-formed programs

$$\text{WP}\ \frac{\mathsf{k} \in \mathsf{K} \implies \mathsf{K} \vdash_s \mathsf{k}\ \checkmark \qquad \Gamma\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t}}{\mathsf{e}\ \mathsf{K}\ \checkmark_s}$$

$\boxed{\sigma \vdash_s\ \mathsf{K}\ \checkmark \mathbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \}}$ Well-formed classes

$$\text{WCL}\ \frac{\begin{array}{c}\mathsf{nodups}(\mathsf{fd}_1, \mathsf{md}_1..) \\ \mathsf{fd} \in \mathsf{fd}_1.. \implies \mathsf{K} \vdash_s \mathsf{fd}\ \checkmark \\ \mathsf{md} \in \mathsf{md}_1.. \implies \mathsf{this} : \mathsf{C}\ \ \mathsf{K} \vdash_s\ \mathsf{md}\ \checkmark\end{array}}{\mathsf{K} \vdash_s\ \mathbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \}\ \checkmark}$$

$\boxed{\Gamma\ \sigma \vdash_s\ \mathsf{K}\ \checkmark\ \mathsf{md}}$ Well-formed methods

$$\text{WT}\ \frac{\begin{array}{c}\Gamma\ \mathsf{x} : \mathsf{C}\ \ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{D} \\ \mathsf{K} \vdash_s\ \mathsf{C}\ \checkmark \qquad \mathsf{K} \vdash_s\ \mathsf{D}\ \checkmark\end{array}}{\Gamma\ \mathsf{K} \vdash_s\ \mathsf{m}(\mathsf{x} : \mathsf{C}) : \mathsf{D}\ \{\mathsf{e}\}\ \checkmark}$$

$$\text{WWT}\ \frac{\begin{array}{c}\Gamma\ \mathsf{x} : \mathsf{t}\ \ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t}' \qquad \mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark \\ \mathsf{K} \vdash_s\ \mathsf{t}'\ \checkmark \qquad \mathsf{kty}(\mathsf{t}) = \mathsf{kty}(\mathsf{t}') = \star\end{array}}{\Gamma\ \mathsf{K} \vdash_s\ \mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}'\ \{\mathsf{e}\}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark}$ Well-formed types

$\boxed{\mathsf{K} \vdash_s\ \mathsf{f} : \mathsf{t}\ \checkmark}$ Well-formed fields

$$\text{WF}\ \frac{\mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark}{\mathsf{K} \vdash_s\ \mathsf{f} : \mathsf{t}\ \checkmark}$$

$$\text{WA}\ \frac{}{\mathsf{K} \vdash_s\ \star\ \checkmark} \qquad \text{WC}\ \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_s\ \mathsf{C}\ \checkmark}$$

$$\text{WW}\ \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_s\ ?\mathsf{C}\ \checkmark}$$

Well-formedness for Behavioral, Optional, Transient

The well-formedness judgments for Behavioral, Optional, and Transient is a subset of the well-formedness judgment for KafKa.

$\boxed{\mathsf{e\ K}\ \checkmark_x}$ Well-formed programs $\boxed{\sigma\ \mathsf{K} \vdash_s\ \mathbf{class}\ \mathsf{C}\{\ \mathsf{fd}_1..\ \mathsf{md}_1..\}\ \checkmark}$ Well-formed classes

SWF-PROG

$$\frac{\Gamma\ \cdot\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t} \qquad \mathsf{k} \in \mathsf{K} \implies \cdot\mathsf{K} \vdash_s\ \mathsf{k}\ \checkmark}{\mathsf{e\ K}\ \checkmark_s}$$

SWF-CLASS

$$\frac{\mathsf{nodups}(\mathsf{fd}_1.., \mathsf{md}_1..) \qquad \mathsf{fd} \in \mathsf{fd}_1.. \implies \mathsf{K} \vdash_s\ \mathsf{fd}\ \checkmark \qquad \mathsf{md} \in \mathsf{md}_1.. \implies \mathsf{this} : \mathsf{C}\ \ \mathsf{K} \vdash_s\ \mathsf{md}\ \checkmark}{\mathsf{K} \vdash_s\ \mathbf{class}\ \mathsf{C}\{\ \mathsf{fd}_1..\ \mathsf{md}_1..\}\ \checkmark}$$

$\boxed{\Gamma\ \mathsf{K} \vdash_s\ \mathsf{md}\ \checkmark}$ Well-formed methods

SWF-TYMETH

$$\frac{\Gamma\ \mathsf{x} : \mathsf{C}\ \ \mathsf{K e} \vdash_s \mathsf{D} : \qquad \mathsf{K} \vdash_s\ \mathsf{C}\ \checkmark \qquad \mathsf{K} \vdash_s\ \mathsf{D}\ \checkmark}{\Gamma\ \mathsf{K} \vdash_s\ \mathsf{m}(\mathsf{x} : \mathsf{C}) : \mathsf{D}\ \{\mathsf{e}\}\ \checkmark}$$

SWF-DYMETH

$$\frac{\Gamma\ \mathsf{x} : \star\ \ \mathsf{K e} \vdash_s \star : \qquad \mathsf{K} \vdash_s\ \star\ \checkmark \qquad \mathsf{K} \vdash_s\ \star\ \checkmark}{\Gamma\ \mathsf{K} \vdash_s\ \mathsf{m}(\mathsf{x} : \star) : \star\ \{\mathsf{e}\}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash_s\ \mathsf{f} : \mathsf{t}\ \checkmark}$ Well-formed fields

SWF-FIELD

$$\frac{\mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark}{\mathsf{K} \vdash_s\ \mathsf{f} : \mathsf{t}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark}$ Well-formed types

SWT-ANY

$$\frac{}{\mathsf{K} \vdash_s\ \star\ \checkmark}$$

SWT-TYPE

$$\frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_s\ \mathsf{C}\ \checkmark}$$

C Source semantics type systems and translations

To avoid unnecessary clutter, we represent the source languages using the common syntax reported in Fig. 20. This defines a simple object calculus similar to KafKa, but without method overloading and, most importantly, cast operations. Lastly, we also give a source-level type system for each language (notated $\vdash_s$), which are largely identical. To simplify the presentation, we will elide the identical rules, instead presenting only the altered rules.

$$
\begin{array}{rcl}
\mathsf{e} & ::= & \mathsf{x} \mid \mathsf{this} \mid \mathsf{this.f} \\
& & \mid \mathsf{this.f} = \mathsf{e} \mid \mathsf{e.m(e)} \\
& & \mid \mathsf{that} \mid \mathbf{new}\ \mathsf{C(e_1..)}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{k} & ::= & \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{fd_1..}\ \mathsf{md_1..}\,\} \\
\mathsf{t} & ::= & \star \mid \mathsf{C} \\
\mathsf{md} & ::= & \mathsf{m(x:t):t}\ \{\mathsf{e}\} \\
\mathsf{fd} & ::= & \mathsf{f:t}
\end{array}
$$

Figure 20 Common syntax of source languages.

C.1 Optional

Our formalization of TypeScript's type system is in Fig. 21. The type system relies on the *convertibility* relation, denoted $\mathsf{K} \vdash_s \mathsf{t} \Rrightarrow \mathsf{t}'$, which captures precisely the implicit type conversions allowed by TypeScript. The convertible relation appears in Fig. 22 and states that a type is convertible to a super type and that $\star$ is convertible to anything and conversely.

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

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

Figure 21 Optional type system.

The TypeScript compiler translates code to JavaScript with all types erased. Since convertibility allows arbitrary values can be passed whenever a $\star$ value is expected, method calls may fail because the receiver need not have the requested method. The designers of TypeScript saw this unsoundness as a way to ensure that types do not get in the way of running correct programs, e.g. when importing a new library with type annotations inconsistent with existing client code; and an insurance for backwards compatibility, as ignoring types means all browsers can run TypeScript code – with no additional overhead.

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

Figure 22 Optional type convertibility.

Observe that the optional translations, in Fig. 23, does insert some structural casts to $\star$, they are needed for the result to be well-typed, but these have no operational effect (a structural cast to $\star$ always succeed at runtime). The unsoundness of the optional type system is evidence in the translation, by discarding the type of the callee and systematically relying on dynamic method invocation for method calls it clear that optional programs can get stuck at any point of their execution.

$$[\![\textbf{class C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] = \textbf{class C}\,\{\,\mathsf{fd}'_1..\ \mathsf{md}'_1..\,\}$$
$$\textbf{where}\quad \mathsf{fd}'_1 = \mathsf{f}:\star\ ..\quad \mathsf{fd}_1 = \mathsf{f}:\mathsf{t}..\quad \mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}_1):\mathsf{t}_2\ \{\mathsf{e}\}$$
$$\mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\star):\star\ \{\mathsf{e}'\}\ ..\quad \mathsf{e}' = [\![\mathsf{e}]\!]$$

$$\begin{array}{lll}
[\![\mathsf{this.f}]\!] & = & \mathsf{this.f} \\
[\![\mathsf{this.f} = \mathsf{e}]\!] & = & \mathsf{this.f} = \mathsf{e}' \qquad \textbf{where}\ \ \mathsf{e}' = [\![\mathsf{e}]\!] \\
[\![\mathsf{this}]\!] & = & <\star>\,\mathsf{this} \\
[\![\mathsf{x}]\!] & = & \mathsf{x} \\
[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!] & = & \mathsf{e}'_1@\mathsf{m}_{\star\to\star}(\mathsf{e}'_2) \qquad \textbf{where}\ \ \mathsf{e}'_1 = [\![\mathsf{e}_1]\!]\ \ \mathsf{e}'_2 = [\![\mathsf{e}_2]\!] \\
[\![\textbf{new C}(\mathsf{e}_1..)]\!] & = & <\star>\,\textbf{new C}(\mathsf{e}'_1..) \qquad \textbf{where}\ \ \mathsf{e}'_1 = [\![\mathsf{e}_1]\!]\ ..
\end{array}$$

Figure 23 Optional translation.

C.2 Concrete

The formalization of the Thorn type system is built on top of the rules presented for TypeScript in Fig. 22 and Fig. 21. The definition of subtyping is extended to account for optional types, this appears in Fig. 24. Optional types are subtypes if the corresponding concrete types are subtypes. Concrete types are subtypes of optional types, if the relation holds on concrete types. The convertibility rule must be extended by one case, as shown in Fig. 25, this extra rule states that an optional type is convertible to a concrete parent type. Type rule for method calls, shown in Fig. 26 must be extended to handle receivers of optional types, they are treated as if they were concrete types.

$$\frac{\mathsf{M}\ \mathsf{K} \vdash_s \mathsf{C} <: \mathsf{D}}{\mathsf{M}\ \mathsf{K} \vdash_s ?\mathsf{C} <: ?\mathsf{D}} \qquad \frac{\mathsf{M}\ \mathsf{K} \vdash_s \mathsf{C} <: \mathsf{D}}{\mathsf{M}\ \mathsf{K} \vdash_s \mathsf{C} <: ?\mathsf{D}}$$

Figure 24 Concrete subtyping.

$$\frac{\cdot\ \mathsf{K} \vdash_s \mathsf{C} <: \mathsf{D}}{\mathsf{K} \vdash_s ?\mathsf{C} \Rightarrow \mathsf{D}}$$

Figure 25 Concrete type convertibility.

$$\frac{\Gamma\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t}' \qquad (\mathsf{t}' = \mathsf{C} \quad \vee \quad \mathsf{t}' = ?\mathsf{C}) \qquad \mathsf{m}(\mathsf{t}_1) : \mathsf{t}_2 \in \mathsf{K}(\mathsf{C}) \qquad \Gamma\ \mathsf{K} \vdash_s \mathsf{e}' : \mathsf{t}'' \qquad \mathsf{K} \vdash_s \mathsf{t}'' \Rightarrow \mathsf{t}_1}{\Gamma\ \mathsf{K} \vdash_s \mathsf{e}.\mathsf{m}(\mathsf{e}') : \mathsf{t}_2}$$

Figure 26 Concrete type system.

The translation of Thorn into KafKa is given in Fig. 27. As with TypeScript translation proceeds top-down. The differences are that the translation function for expressions, $[\![\mathsf{e}]\!]_\Gamma$, takes a source-level typing environment as input. This is used to record the expect type of this and arguments x to methods. Furthermore, the translation can also request the insertion of casts, this is done with the function $(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma$ which translates an expression and ensures that the result is of type t. The most interesting case in the translation is the handling of method invocation $\mathsf{e}.\mathsf{m}(\mathsf{e}')$. If the type of e is a concrete C, then a statically resolved invocation of the form $\mathsf{e}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$ will be emitted. If e is dynamic or an optional type, then a dynamically resolved call of the form $\mathsf{e}@\mathsf{m}_{\star\to\star}(\mathsf{e}')$ is emitted. The argument of statically resolved method invocation, as well as constructors and field assignment are all translated using the auxiliary function as their expected type is known. This function translates its argument, checks if its type is a subtype of the expected type t, and if not it inserts the appropriate cast. The cast is performed in the KafKa type system, and not in the source type system, and must respect the mapping from Thorn types to KafKa types. This mapping is defined by the $\mathsf{kty}(\mathsf{t})$ function: $\mathsf{kty}(\mathsf{t}) = \star$ if $\mathsf{t} = ?\mathsf{C}$ or $\mathsf{t} = \star$, t otherwise. Thorn optional and dynamic types are mapped to the $\star$ type, while concrete types are unchanged.

$$[\![\textbf{class C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] = \textbf{class C}\,\{\,\mathsf{fd}'_1..\ \mathsf{md}'_1..\ \mathsf{md}''_1..\,\}\quad \textbf{where}$$

$$\mathsf{fd}'_1 = \mathsf{f}:\mathsf{kty}(\mathsf{t})\ ..\qquad \mathsf{fd}_1 = \mathsf{f}:\mathsf{t}..$$

$$\mathsf{md}'_1 = \mathsf{m}(\mathsf{x}:\mathsf{kty}(\mathsf{t}_1)):\mathsf{kty}(\mathsf{t}_2)\ \{\mathsf{e}'\}\ ..$$

$$\mathsf{md}_1 = \mathsf{m}(\mathsf{x}:\mathsf{t}_1):\mathsf{t}_2\ \{\mathsf{e}\}..\quad \mathsf{e}' = (\!|\mathsf{e}|\!)^{\mathsf{t}_2}_{\mathsf{this}\,\mathsf{C}\times\mathsf{t}_1}\ ..$$

$$\mathsf{md}''_1 = \mathsf{m}(\mathsf{x}:\star):\star\ \{<\star>\mathsf{this}.\mathsf{m}_{\mathsf{t}_1\to\mathsf{t}_2}(<\mathsf{kty}(\mathsf{t}_1)>\mathsf{x})\}$$
$$\textbf{if}\ \ \mathsf{kty}(\mathsf{t}_1)=\mathsf{D}\ \ \textbf{or}\ \ \mathsf{kty}(\mathsf{t}_2)=\mathsf{D}$$
$$\text{empty}\quad \textbf{otherwise}\ ..$$

$$[\![\mathsf{x}]\!]_\Gamma = \mathsf{x}$$

$$[\![\mathsf{this.f}]\!]_\Gamma = \mathsf{this.f}$$

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

$$[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}'_1@\mathsf{m}_{\star\to\star}(\mathsf{e}'_2)\quad \textbf{where}\quad \mathsf{K},\Gamma\vdash\mathsf{e}_1:\mathsf{t}\quad \mathsf{kty}(\mathsf{t})=\star\quad \mathsf{e}'_1=[\![\mathsf{e}_1]\!]_\Gamma$$
$$\mathsf{e}'_2=(\!|\mathsf{e}_2|\!)^\star_\Gamma$$

$$[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}'_1.\mathsf{m}_{\mathsf{t}_2\to\mathsf{t}'_2}(\mathsf{e}'_2)\quad \textbf{where}\quad \mathsf{K},\Gamma\vdash\mathsf{e}_1:\mathsf{C}\quad \mathsf{e}'_1=[\![\mathsf{e}_1]\!]_\Gamma\quad \mathsf{e}'_2=(\!|\mathsf{e}_2|\!)^{\mathsf{t}_2}_\Gamma$$
$$\mathsf{m}(\mathsf{t}_1):\mathsf{t}'_1\in\mathsf{K}(\mathsf{C})\ \mathsf{t}_2=\mathsf{kty}(\mathsf{t}_1)\quad \mathsf{t}'_2=\mathsf{kty}(\mathsf{t}'_1)$$

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

$$(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = \mathsf{e}'\quad \textbf{where}\quad \mathsf{K},\Gamma\vdash\mathsf{e}:\mathsf{t}'\quad \mathsf{e}'=[\![\mathsf{e}]\!]_\Gamma$$
$$\mathsf{K}\vdash\mathsf{kty}(\mathsf{t}')<:\mathsf{kty}(\mathsf{t})$$

$$(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = <\mathsf{kty}(\mathsf{t})>\mathsf{e}'\quad \textbf{where}\quad \mathsf{K},\Gamma\vdash\mathsf{e}:\mathsf{t}'\quad \mathsf{e}'=[\![\mathsf{e}]\!]_\Gamma$$
$$\mathsf{K}\vdash\mathsf{kty}(\mathsf{t}')\not<:\mathsf{kty}(\mathsf{t})$$

Figure 27 Concrete translation.

C.3 Behavioral

The translation for Typed Racket, shown in Fig. 28 maps classes to homonymous classes and types to types of the same name. The $\mathsf{kty}(t)$ function is the identity. Calls with a receiver of type $\star$ are translated to KafKa dynamic calls, and calls with a receiver of some type C are translated to statically typed calls. The auxiliary type-directed translation function $(\!|e|\!)^{t}_{\Gamma}$ introduces behavioral casts to $\star$ or C appropriately. Thus casts can appear are at typed/untyped boundaries in assignment, argument of calls or constructors. Because of the strong guarantee provided by the behavioral cast, the Typed Racket translation is straightforward. In Typed Racket, every typed value is assumed to behave as specified, delegating the complexity of checking for consistency with the type to the wrapper introduced by the behavioral cast. So the difference with the TypeScript translation is that typed code is able to use typed accesses. The difference with Thorn is essentially the presence of wrappers and the fact that each wrapper application only check the presence of methods names rather than complete signatures for concrete types.

C.4 Transient

The Transient static type system is based on TypeScript except that the convertibility rules now builds on an auxiliary *consistency* relation, defined in Fig. 30 to relate types t and t′. The modified convertibility rule appears in Fig. 29. Consistent subtyping holds between types with signatures that agree up to $\star$. It is worth observing that the Transient runtime does not use consistent subtyping, instead it merely validates that all required method names are present.

$$
\begin{array}{lll}
[\![\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}\colon \mathsf{t})\colon \mathsf{t}'\ \{\mathsf{e}'_1\}\ .. \\
 & & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\mathsf{e}_1\}\ .. \\
 & & \mathsf{e}'_1 = [\![\mathsf{e}_1]\!]_{\mathsf{x}:\mathsf{t}\ \mathsf{this}:\mathsf{C}} \\
[\![\mathsf{x}]\!]_\Gamma = \mathsf{x} & & \\
[\![\mathsf{this.f}]\!]_\Gamma = \mathsf{this.f} & & \\
[\![\mathsf{this.f} = \mathsf{e}]\!]_\Gamma = \mathsf{this.f} = \mathsf{e}' & \mathbf{where} & \mathsf{K} \vdash \mathsf{this} : \mathsf{C} \quad \mathsf{e}' = (\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma \quad \mathsf{f}:\mathsf{t} \in \mathsf{K}(\mathsf{C}) \\
[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}'_1@\mathsf{m}_{\star\to\star}(\mathsf{e}'_2) & \mathbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \star \quad \mathsf{e}'_1 = [\![\mathsf{e}_1]\!]_\Gamma \quad \mathsf{e}'_2 = (\!|\mathsf{e}_2|\!)^{\star}_\Gamma \\
[\![\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)]\!]_\Gamma = \mathsf{e}'_1.\mathsf{m}_{\mathsf{D}_1\to\mathsf{D}_2}(\mathsf{e}'_2) & \mathbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \mathsf{C} \quad \mathsf{e}'_1 = [\![\mathsf{e}_1]\!]_\Gamma \quad \mathsf{e}'_2 = (\!|\mathsf{e}_2|\!)^{\mathsf{D}_1}_\Gamma \\
 & & \mathsf{m}(\mathsf{D}_1)\colon \mathsf{D}_2 \in \mathsf{K}(\mathsf{C}) \\
[\![\mathbf{new}\ \mathsf{C}(\mathsf{e}_1..)]\!]_\Gamma = \mathbf{new}\ \mathsf{C}(\mathsf{e}'_1..) & \mathbf{where} & \mathsf{e}'_1 = (\!|\mathsf{e}_1|\!)^{\mathsf{t}_1}_\Gamma\ ..\ \mathsf{f}_1 : \mathsf{t}_1 \in \mathsf{K}(\mathsf{C})\ .. \\
(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = \mathsf{e}' & \mathbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}' \quad \mathsf{e}' = [\![\mathsf{e}]\!]_\Gamma \\
(\!|\mathsf{e}|\!)^{\mathsf{t}}_\Gamma = \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} & \mathbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' \quad \mathsf{K} \vdash \mathsf{t} \not<: \mathsf{t}' \quad \mathsf{e}' = [\![\mathsf{e}]\!]_\Gamma
\end{array}
$$

Figure 28 Behavioral translation.

$$\frac{\cdot\ K \vdash t \lesssim t'}{K \vdash_s t \Rightarrow t'}$$

Figure 29 Transient convertibility.

$$\frac{}{M\ K \vdash \star \lesssim C} \qquad \frac{}{M\ K \vdash C \lesssim \star} \qquad \frac{}{M\ K \vdash t \lesssim t} \qquad \frac{C <: D \in M}{M\ K \vdash C \lesssim D}$$

$$\frac{mt \in K(D) \implies mt' \in K(C) \quad \wedge \quad M'\ K \vdash mt \lesssim mt' \qquad M' = M\ C <: D}{M\ K \vdash C \lesssim D} \qquad \frac{M\ K \vdash t_2 \lesssim t_1 \qquad M\ K \vdash t_1' \lesssim t_2'}{M\ K \vdash m(t_1) : t_1' \lesssim m(t_2) : t_2'}$$

Figure 30 Transient consistent subtyping.

$$[\![\textbf{class } C\,\{\,fd_1..\ md_1..\,\}]\!] = \textbf{class } C\,\{\,fd'_1..\ md'_1..\,\}$$
$$\textbf{where}\quad fd'_1 = f:\star\ ..\quad fd_1 = f:t..\quad md'_1 = m(x:\star):\star\,\{<t>x\ ;\ e'_1\}\ ..$$
$$md_1 = m(x:t):t'\,\{e\}..\quad e'_1 = (\!|e|\!)^{t'}_{x:t\ this:C}\ ..$$

$[\![this]\!]_\Gamma$	$=$	$this$		
$[\![x]\!]_\Gamma$	$=$	$<t>x$	**where**	$K,\Gamma \vdash x : t$
$[\![this.f]\!]_\Gamma$	$=$	$<t>this.f$	**where**	$K,\Gamma \vdash this : C \quad f:t \in K(C)$
$[\![this.f = e]\!]_\Gamma$	$=$	$<t>this.f = e'$	**where**	$K,\Gamma \vdash this : C \quad f:t \in K(C)$ $e' = (\!|e|\!)^\star_\Gamma$
$[\![e_1.m(e_2)]\!]_\Gamma$	$=$	$e'_1@m_{\star\to\star}(e'_2)$	**where**	$K,\Gamma \vdash e_1 : \star \quad e'_1 = [\![e_1]\!]_\Gamma$ $e'_2 = (\!|e_2|\!)^\star_\Gamma$
$[\![e_1.m(e_2)]\!]_\Gamma$	$=$	$<t'>e'_1.m_{\star\to\star}(e'_2)$	**where**	$K,\Gamma \vdash e_1 : C \quad m(t):t' \in K(C)$ $e'_1 = [\![e_1]\!]_\Gamma \quad e'_2 = (\!|e_2|\!)^\star_\Gamma$
$[\![\textbf{new } C(e_1..)]\!]_\Gamma$	$=$	$\textbf{new } C(e'_1..)$	**where**	$f_1:t_1 \in K(C) \quad e'_1 = (\!|e_1|\!)^\star_\Gamma\ ..$
$(\!|e|\!)^t_\Gamma$	$=$	$<t>e'$	**where**	$K,\Gamma \vdash e : t' \quad K \vdash t \lesssim t'$ $K \vdash t \not<: t' \quad e' = [\![e]\!]_\Gamma$
$(\!|e|\!)^t_\Gamma$	$=$	e'	**where**	$K,\Gamma \vdash e : t' \quad K \vdash t <: t'$ $e' = [\![e]\!]_\Gamma$

Figure 31 Transient translation.

The Transient translation appears in Fig. 31. Each class is translated to a homonymous KafKa class, field types are translated to $\star$, method types are given the $\star \to \star$ signature. Since argument and return types are erased, our translation can use structural casts to implement Transient runtime checks. They degenerate to simple inclusion checks on method names. The translation of method invocation makes explicit the Transient guarantee. A method call $e.m(e')$ is translated to $e.m_{\star\to\star}(e')$ if the type e is not $\star$. This translation combined with the soundness of KafKa entails that the method call will not get stuck. Of course of e is of type $\star$, the call will be translated to $e@m_{\star\to\star}(e')$ which can get stuck. To achieve that guarantee the translation must insert structural casts at every expression read. Transient also checks arguments of methods, since the expression may not use the argument (but Transient checks it anyway) our translation generates method bodies of the form $<t>x; e$ where e is the body of the expression and the semi colon is syntactic sugar for sequencing. Likewise, in order to ensure that field assignment will not get stuck, Transient gives all fields type $\star$, then checks the field value on reads based on the static typing.

D Litmus tests

Below we present source code for each of the litmus tests of Fig. 8 of section 4.

Concrete

The code for the litmus tests in Thorn.

Litmus Test 1:

```
class A() { def m(x:A):A = this; }
class I() { def n(x:I):I = this; }
class T() {
  def s(x:I):T = this;
  def t(x:dyn):dyn = this.s(x);
}
T().t(A());
```

Litmus Test 2:

```
class Q() { def n(x: Q): Q = this;}
class A() { def m(x:A): A = this;}
class I() { def m(x:Q):I = this;}
class T() {
   def s(x:I):T = this;
   def t(x:dyn):dyn = this.s(x);
}
T().t(A());
```

Litmus Test 3:

```
class C() { def m(x:C):C = x; }
class D() { def n(x:D):D = x; }
class E() { def m(x:D):D = x; }
class F() {
   def m(x:E):E = x;
   def n(x:dyn):dyn = this.m(x);
}
F().n(C()).m(C());
```

Optional

The code for the litmus tests in TypeScript.

Litmus Test 1:

```
class A { m(x: A): A { return this } }
class I { n(x:I):I { return this } }
class T {
    s(x: I): T { return this }
    t(x: any): any { return this.s(x) }
}
new T().t(new A())
```

Litmus Test 2:

```
class Q { n(x: Q): Q { return this } }
class A { m(x: A): A { return this } }
class I { m(x:Q):I { return this } }
class T {
    s(x: I): T { return this }
    t(x: any): any { return this.s(x) }
}
new T().t(new A())
```

Litmus Test 3:

```
class C { m(x: C): C { return x } }
class D { n(x: D): D { return x } }
class E { m(x: D): D { return x } }
class F {
    m(x: E): E { return x }
    n(x: any): any { return this.m(x) }
}
new F().n(new C()).m(new C())
```

Behavioral

The code for the litmus tests in Typed Racket.

Litmus Test 1:

```
#lang racket
(module u racket
  (define Tp% (class object%
                (super-new)
                (define/public (t x) (send this s x))))
  (provide Tp%))
(module t typed/racket
  (require/typed (submod ".." u) [Tp% (Class [t (-> Any Any)])])
  (define-type A (Instance (Class (m (-> A A)))))
  (define-type I (Instance (Class (n (-> I I)))))
  (define-type T (Instance (Class (s (-> I T)))))
  (define T% (class Tp%
               (super-new)
               (: s (-> I T))
               (define/public (s x) this)))
  (define A% (class object%
               (super-new)
               (: m (-> A A))
               (define/public (m x) this)))
  (provide T% A%))
(require 't)
(send (new T%) t (new A%))
```

Litmus Test 2:

```
#lang racket
(module u racket
  (define Tp% (class object%
                (super-new)
                (define/public (t x) (send this s x))))
  (provide Tp%))
(module t typed/racket
  (require/typed (submod ".." u) [Tp% (Class [t (-> Any Any)])])
  (define-type Q (Instance (Class (n (-> Q Q)))))
  (define-type A (Instance (Class (m (-> A A)))))
  (define-type I (Instance (Class (m (-> Q I)))))
  (define-type T (Instance (Class (s (-> I T)))))
  (define T% (class Tp%
               (super-new)
               (: s (-> I T))
               (define/public (s x) this)))
  (define A% (class object%
               (super-new)
               (: m (-> A A))
               (define/public (m x) this)))
  (provide T% A%))
(require 't)
(send (new T%) t (new A%))
```

Litmus Test 3:

```
#lang racket
(module u racket
  (define Fp% (class object%
                (super-new)
                (define/public (n x) (send this m x))))
  (provide Fp%))
(module t typed/racket
  (require/typed (submod ".." u) [Fp% (Class [n (-> Any Any)])])
  (define-type C (Instance (Class (m (-> C C)))))
  (define-type E (Instance (Class (m (-> D D)))))
  (define-type D (Instance (Class (n (-> D D)))))
  (define F% (class Fp%
               (super-new)
               (: m (-> E E))
               (define/public (m x) x)))
  (define C% (class object%
               (super-new)
               (: n (-> C C))
               (define/public (n x) x)))
  (provide F% C%))
(require 't)
(send (send (new F%) n (new C%)) m (new C%))
```

Transient

The code for the litmus tests in Transient Reticulated Python.

Litmus Test 1:

```
class A:
   def m(self, x:A) -> A:
    return self
class I:
   def n(self, x:I) -> I:
     return self
class T:
   def s(self, x:I) -> T:
     return self
   def t(self, x:Dyn) -> Dyn:
     return self.s(x)
T().t(A())
```

Litmus Test 2:

```
class C:
   def n(self, x:C) -> C:
     return self
class Q:
   def m(self, x:Q) -> Q:
     return self
class A:
   def m(self, x:A) -> A:
     return self
class I:
   def m(self, x:Q) -> I:
     return self
class T:
   def s(self, x:I) -> T:
     return self
   def t(self, x:Dyn) -> Dyn:
     return self.s(x)
T().t(A())
```

Litmus Test 3:

```
class C:
  def m(self, x:C) -> C:
```

```
     return x
class D:
  def n(self, x:D) -> D:
     return x
class E:
  def m(self, x:D) -> D:
     return x
class F:
  def m(self, x:E) -> E:
     return x
  def n(self, x:Dyn) -> Dyn:
     return self.m(x)
F().n(C()).m(C())
```