Gradual Types for Objects Redux

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

Northeastern University, CVUT and INRIA

Abstract. The enduring popularity of dynamically typed languages has given rise to a cottage industry of static type systems, often called *gradual type systems*, that let developers annotate legacy code piecemeal. Type soundness for a program which mixes typed and untyped code does not ensure the absence of errors at runtime, rather it means that some errors will caught at type checking time, while other will be caught as the program executes. After a decade of research it is clear that the combination of mutable state, self references and subtyping presents interesting challenges to designers of gradual type systems. This paper reviews the state of the art in gradual typing for objects. We introduce KafKa, a class-based object calculus with a static type system, dynamic method dispatch, transparent wrappers and dynamic class generation. We model key features of four gradual type systems by translation to KafKa and discuss the implications of the respective designs.

1 Introduction

A decade ago Siek and Taha [1] presented a gradual type system for a variant of Abadi and Cardelli's object-based calculus [2]. 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 [3].

A decade later, practical realizations of Siek and Taha's elegant idea have proved elusive. One potential reason may be that the original paper did not consider state. The combination of mutable state, aliasing and subtyping prevalent in object-oriented languages complicates enforcement strategies, as one must consider situations where an object is being accessed and mutated at different types. While several solutions have been proposed, their performance implications appear daunting. Predictably, developers of industrial languages have chosen to compromise on soundness to avoid degrading performance.

This paper explores the design space of gradual type systems for object-oriented languages by studying four distinctive systems, idealizations of TypeScript [4], Thorn [5], Typed Racket [6], and Transient Reticulated Python [7], which significantly different semantics. Beyond the difference in runtime overheads imposed by these systems, which we do not study here, there is not even agreement over what constitutes a runtime error. To highlight the issue, we propose a litmus test consisting of three simple programs, well-typed in the four respective source languages, which allow us to distinguish between the type

systems. In an untyped language these programs run without error, but with gradual types we observe different numbers of errors depending on the runtime enforcement strategy used.

Our aim is to shed light on core features of each type system, namely their treatment of objects, and which errors are caught statically and which ones are caught dynamically. These four languages were selected because they had open source implementations and their designs were representative of different philosophies. TypeScript stands for unsound systems that work by erasure, such as Dart and Hack. Thorn combines dynamic and static types inspired by C# and further developed in StrongScript. Typed Racket provides a sound type system that is close in spirit to Siek and Taha's original idea. Lastly, Transient was designed to decrease the overhead of soundness, while retaining some systematic runtime checks.

To capture the essence of gradual typing, we present translations of representative subsets of four gradually typed languages into a common target. This target language, dubbed KafKa, is modeled on the features exposed by the intermediate languages of the Java Virtual Machine and the Common Language Runtime – both of which have been used as targets for implementations of gradual typing systems. The particular features we exploit are a simple static type system equipped with dynamic casts that check the runtime type of values, the ability to dynamically resolve method targets, and support for generating new classes at runtime.

Translating a gradually typed language into a statically typed language, such as the JVM's bytecode or KafKa makes explicit which of the underlying type system's guarantees can be relied upon statically, and where dynamism is really required. Our core contribution is thus four translations, each taking a gradually typed expression $\mathsf{e} : \mathsf{T}$ in the source language and returning the corresponding KafKa term. The translation of a term entails translation of the classes needed to evaluate it. The translation allows us to study where program can get stuck in the different systems. With gradual types, an expression $\mathsf{x.m(e)}$, where x is declared to be of class C, can have significantly different behavior depending on choices made while designing the type system. TypeScript has *optional types*; any call can get stuck, as every type translates to $\star$. Thorn has *concrete types*; a program will not get stuck at calls on these types as they translate to statically typed method calls in KafKa. Typed Racket has *promised types*; a well-typed program will not get stuck at a call to m, because x refers to an object, or wrapper, that implements m. Wrappers may fail if their target does not behave like the type that they enforce. Transient Python will also allow calls to go through, but may get stuck if the called function returns a value of the wrong type.

The design of KafKa is another contribution. KafKa is a statically typed object calculus. It is class based (with an explicit class table K) with mutable state (a heap address a refers to an object with a set of private fields denoted by f). It allows the dynamic generation of wrapper classes (by addition of new classes to the class table K and allocation of objects a). Methods can be statically resolved, denoted by a typed call $\mathsf{a.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{x})$, or can be dynamically resolved, denoted by

a dynamic call $\mathsf{a@m}_{\star\to\star}(\mathsf{x})$. The KafKa type system has two kinds of types; classes which give rise to homonymous types, and the dynamic type $\star$. KafKa subtyping is structural with recursion. The heart of any gradual type system are the explicit casts that are inserted at type boundaries. Structural casts, $<\mathsf{t}>\ \mathsf{a}$, check if the object referenced by a is a subtype of type t. Behavioral casts, $\blacktriangleleft\ \mathsf{t}\ \blacktriangleright\ \mathsf{a}$, create wrappers that enforce a specific type, this entails runtime generation of a new class. Finally, We give KafKa an operational semantics. We report on a mechanized proof of soundness for KafKa inclusive of its casts, as well as a proof-of-concept implementation of the calculus on top of the .NET virtual machine.

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 [8], which led to Bracha's notion of pluggable types [9]. 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 words of Richards *et al.* [10], an optional type system is *trace preserving*, which means that if a 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 [11], TypeScript [4] and Dart [12].

On the functional side, the ancestry is dominated by the work of Felleisen and his students. The Typed Scheme [6] design that later became Typed Racket is influenced by the authors' earlier work on higher-order contracts [13]. 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 promised 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 remains 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 [3] 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 [1] 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 [7] 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.

Other noteworthy systems include Gradualtalk [14], C# 4.0 [15], Thorn [5], and StrongScript [10]. 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. 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. Most languages base subtyping on explicit name-based subtype declarations, 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 [10]. StongScript extends TypeScript changing subtyping back to nominal. The consistency relation used in Reticulated Python is fundamentally structural, it would be nonsensical to use it in a nominal system.[1] For Racket, the heavy use of first-class classes and class generation naturally leads to structural subtyping as many of the classes being manipulated have no names. Optional types are the default execution mode for Dart, Hack and TypeScript. Transient 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 [7]. 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. Generative casts are used when information such as a type or a blame label must be associated with a reference or an object.

[1] 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.

	Nominal	*Optional types*	*Concrete types*	*Promised 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

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

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.

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 generative casts tend to suffer prohibitive slow downs in pathological cases.

3 Litmus test

One of the salient implications of the various runtime type enforcement strategies used in the four gradual type systems under study is that they have different notions of what constitutes as an erroneous program. In order for a type system to be usable, the reason behind an error needs to be clear.

In this section, we propose a litmus test that can be used to distinguish the four type systems. The three programs of Fig. 2 are statically well-typed in all four gradual type systems. 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. For completeness we provide source code in TypeScript, Typed Racket, Thorn and Reticulated Python in the appendix.

TODO: Rework figure in terms of semantic names

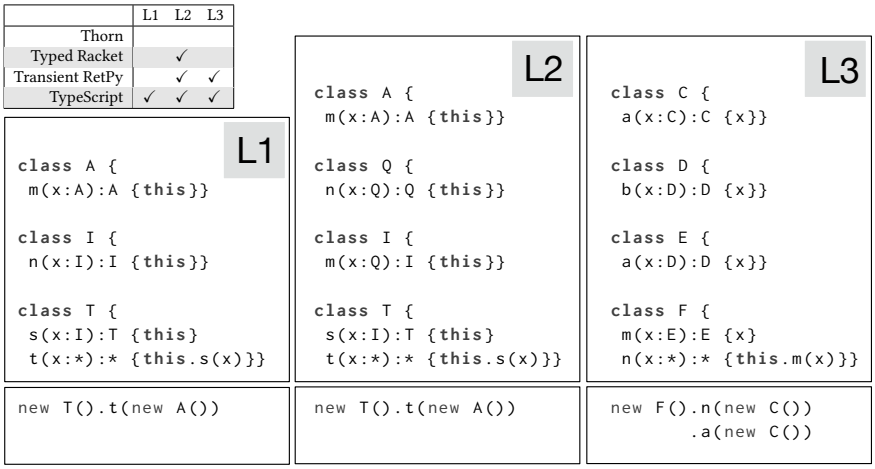

Fig. 2. Gradual typing semantic litmus test.

The programs of Fig. 2 are given in our idealized syntax, presented in Section 5, we assume they are translated to KafKa and that they execute according to the semantics of Section 4. Each program consist of a class table (larger box), and a top-level expression (smaller box).

All programs execute correctly under the optional semantics, as the typing pathologies are never relied upon dynamically.

L1 fails in every sound semantics. Under the concrete semantics, the method call to `this.s(x)` in method `t` of class `T` fails because A is not a subtype of I, as method `m` of class A exhibits different types to method `n` of class I. Similarly, in the behavioral and transient semantics, A and I are also not considered subtypes because their respective class have methods with different sets of names.

L2 fails only under the concrete semantics. As shown in **L1**, concrete type checking requires A to be a subtype of I, while in **L2**, Q is not a subtype of A as methods `m` and `n` exhibit different types. The program works fine under the behavioral and transient semantics because A have the same methods as I.

L3 fails with the concrete and behavioral semantics. Using concrete semantics, the method invocation of `this.m(x)` in method `n` of class F fails because an instance of C is being casted to E, despite C not being a subtype of E. Just as A is not a subtype of I for **L1** and **L2**, C is not a subtype of E. **L3** also fails with the behavioral semantics as the method invocation of .a(new C()) in the top-level expression is calling method a of class E. This requires casting an instance of C to D, which in turn requires method a of class C to only take and return values of type D, however, there are no subtyping relation between C and D. Transient, in contrast, can run **L3** without error, as, unlike Thorn, it does not compare argument types, and, unlike Racket, it does not recall the prior cast to E. Thus given these three programs and the ability to observe errors, one can divine which semantics a language implements.

4 KafKa: A Core Calculus

The basis of our formal approach is KafKa, a class-based, statically typed object-oriented language. The distinctive features of the calculus are its support for both typed and untyped method innovation, as well as that dynamic class generation by 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 have typed intermediate language with support for untyped method call (through their reflection API) and class generation (by dynamic loading). Our design is guided by the intuition that the semantics of object-oriented languages with gradual types can be translated to a common representation by the introduction of *casts* and *wrappers*.

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

Fig. 3. KafKa Syntax.

Syntax KafKa is an object calculus that satisfies the above design requirement. Its syntax is given in Fig. 3. Types consist of class names C, D and the dynamic type, written $\star$. Class definitions have a class name and (possibly empty) sequences of field and method definitions, **class** C { $\mathsf{fd}_1..$ $\mathsf{md}_1..$ }. Field definitions

consist of a field and its type, $\mathsf{f} : \mathsf{t}$. Method definitions have (for simplicity) a single argument and an expression, denoted $\mathsf{m}(\mathsf{x}\colon \mathsf{t}) : \mathsf{t}\ \{\mathsf{e}\}$. 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 $\mathsf{this.f}$ and writing a field is denoted $\mathsf{this.f} = \mathsf{e}$. The calculus supports both statically and dynamically resolved method invocation. A statically resolved call, denoted $\mathsf{e}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$, 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 $\mathsf{m}(\mathsf{t}) : \mathsf{t}'$. A dynamically resolved call, $\mathsf{e}@\mathsf{m}_{\star\to\star}(\mathsf{e}')$, 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 $\mathsf{C}\{\mathsf{a}_1..\}$.

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, $\mathsf{M}\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$, shown in Fig. 4, allows for recursive structural subtyping: for this the environment M keeps track of the set of subtype relations assumed. 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. Recall fields are hidden from the class type signature, so subtyping is limited to method definitions. Key type rules are in Fig. 5. Method calls use syntactic disambiguation to select between typed and untyped methods (dynamic method resolution uses @). 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, the way their soundness is enforced at runtime is very different.

Dynamic Semantics The small step operational semantics for KafKa appears in Fig. 6. To resolve the this reference in field accesses, the syntax of expressions is extended at runtime with forms

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

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

$$\frac{\mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_1' <: \mathsf{t}_1 \quad \mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_2 <: \mathsf{t}_2'}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1)\colon \mathsf{t}_2\ \{\mathsf{e}\} <: \mathsf{m}(\mathsf{x}\colon \mathsf{t}_1')\colon \mathsf{t}_2'\ \{\mathsf{e}'\}}$$

Fig. 4. KafKa subtyping

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

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

Fig. 5. KafKa static semantics (excerpt)

where a is a reference. The static typing of references (Fig. 5) 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

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

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.

Behavioral Casts A 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 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

K	**new** $C(a_1..)$	σ	$\rightarrow$	K	a'	σ'	**where**	a' fresh	$\sigma' = \sigma[a' \mapsto C\{a_1..\}]$
K	$a.f_i$	σ	$\rightarrow$	K	a_i	σ	**where**	$\sigma(a) = C\{a_1..a_i..\}$	
K	$a.f_i = a'$	σ	$\rightarrow$	K	a'	σ'	**where**	$\sigma(a) = C\{a_1..a_i..\}$	$\sigma' = \sigma[a \mapsto C\{a_1..a'..\}]$
K	$a.m_{t\rightarrow t'}(a')$	σ	$\rightarrow$	K	e'	σ	**where**	$e' = [a/\text{this}\ a'/x]e$	$m(x\colon t_1)\colon t_2\ \{e\} \in K(C)$
								$\sigma(a) = C\{a_1..\}$	$\emptyset\ K \vdash t <\colon t_1$
								$\emptyset\ K \vdash t_2 <\colon t'$	
K	$a@m_{\star\rightarrow\star}(a')$	σ	$\rightarrow$	K	e'	σ	**where**	$e' = [a/\text{this}\ a'/x]e$	$m(x\colon \star)\colon \star\ \{e\} \in K(C)$
								$\sigma(a) = C\{a_1..\}$	
K	$<\star> a$	σ	$\rightarrow$	K	a	σ			
K	$<D> a$	σ	$\rightarrow$	K	a	σ	**where**	$\emptyset\ K \vdash C <\colon D$	$\sigma(a) = C\{a_1..\}$
K	$\blacktriangleleft t \blacktriangleright a$	σ	$\rightarrow$	K′	a'	σ'	**where**	$K'\ a'\ \sigma' = \text{bcast}(a, t, \sigma, K)$	
K	E[e]	σ	$\rightarrow$	K′	E[e′]	σ'	**where**	$K\ e\ \sigma \rightarrow K'\ e'\ \sigma'$	

Fig. 6. KafKa dynamic semantics

class (the wrapper itself). Its dynamic semantics is reported in Fig. 7 and Fig. 8. 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 checks that). This prevents ambiguity in method resolution for wrapped objects. When the target type is ⋆, the cast allows a to act like an instance of ⋆.

One way to understand wrappers is that they 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.

Wrapper class generation is implemented by the W and W⋆ functions. Their first three arguments C, md_1.., 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.

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

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

Fig. 7. Behavioral casts

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

Fig. 8. Wrapper class generation

4.1 Type soundness

We have mechanized the type soundness proof for KafKa in Coq, available at *[link omitted for blind review]*. In Coq syntax, the KafKa type soundness theorem is stated as follows:

```
forall (k:ct) (s:heap) (e:expr) (t:type),
  (*condition 1*) WellFormedState k e s ->
  (*condition 2*) HasType nil s k e t ->
  (*case 1*) (exists a:ref, e = Ref a) \/
  (*case 2*) (exists (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') \/
  (exists E:EvalCtx,
    (*case 3*) (exists (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 /\
       forall x e, ~ (In (Method m x Any Any e) (methods C k)
           )) \/
    (*case 4*) (exists (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 5*) (exists 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))))
```

Our Coq mechanization is identical to our formalism. In particular, $\Gamma\,\sigma\,\mathsf{k} \vdash \mathsf{e} : \mathsf{t}$ is equivalent to `HasType g s k e t` and $\mathsf{K}\,\mathsf{e}\,\sigma\,\checkmark$ is equivalent to `WellFormedState k e s`. 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`. `equivExpr` fills in the hole in `E` with the given expression.

Informally, our 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 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 `exists a : ref, e = Ref a`.

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`).

Case 3 The program could be stuck at a dynamic call expression. T A 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.

Case 4 The program could have gotten stuck on a subtype cast. In an expression of the form `SubCast 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 `t'`.

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

As a result of this soundness theorem, the KafKa type system provides a strong guarantee of progress, where most expressions are guaranteed to step. By providing this guarantee, we localize where KafKa programs could potentially encounter runtime errors, which will allow us to reason easily about where gradually typed programs go wrong under each of the semantics.

4.2 Implementation

We designed KafKa to have a close correspondence to the intermediate languages used by common 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. The implementation is available from our website *[link omitted for blind review]*.

The only substantial challenge in the development relates to one of our earliest design choices: the use of a structural type system. As previously mentioned, structural typing is key for the consistent subtyping used in Reticulated Python, but few virtual machines have built-in support for it. Thus 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.

The goal of our implementation was to validate the choice of features to include in KafKa by showing how they match against those provided in modern, high performance VMs, and for most of KafKa's functionality, the implementation was trivial. What our implementation does not do is provide a picture of the performance of the gradual typing systems. This limitation is due to a wide range of factors, including having none of the commonly-cited performance optimizations, such as threesomes [16], combined with the inherent restrictions on what programs can be reasonably written in KafKa.

5 Translating Gradual Type Systems

We are now ready to give semantics to the four gradual type systems of interest by translating them into KafKa. For each source language, 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 language, highlighting the similarities and differences between them.

To avoid unnecessary clutter, we represent the source languages using the common syntax reported in Fig. 9. This defines a simple object calculus similar to KafKa, but without method overloading and, most importantly, cast operations. Additionally, when modeling the Thorn type system, the additional type ?C is added to the type grammar to denote Thorn *like types*. 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.

$$
\begin{array}{llll}
e & ::= & x \mid \mathsf{this} \mid \mathsf{this.f} \mid \mathsf{this.f} = e \mid \mathsf{e.m(e)} \mid \mathsf{that} \mid \mathbf{new}\ \mathsf{C}(e_1..) & k ::= \mathbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \} \\
 & & & t ::= \star \mid \mathsf{C} \mid \mathsf{?C}
\end{array}
\qquad
\begin{array}{ll}
\mathsf{md} ::= \mathsf{m(x:t):t}\ \{e\} & \\
\mathsf{fd} ::= \mathsf{f:t} &
\end{array}
$$

Fig. 9. Common syntax of source languages

Our source language has no explicit casts, instead silently coercing types at typed-untyped boundaries, allowing code like the following:

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

Here, 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.

By deferring to the translation to provide a soundness guarantee, the source language allows us to implement very different approaches to gradual typing within the same framework. Through the simple invariant that well-typed source terms translate to well-typed KafKa terms, we are able to provide a simple soundness guarantee, and provide some intuition as to where different approaches allow programs to go wrong.

5.1 Optional

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

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

$$\frac{\mathsf{f}_1 : \mathsf{t}_1 .. \in \mathsf{K}(\mathsf{C}) \quad \Gamma\ \mathsf{K} \vdash_s \mathsf{e}_1 : \mathsf{t}'_1 .. \quad \mathsf{K} \vdash_s \mathsf{t}'_1 \Rightarrow \mathsf{t}_1 ..}{\Gamma\ \mathsf{K} \vdash_s \mathbf{new}\ \mathsf{C}(\mathsf{e}_1 ..) : \mathsf{C}}$$

Fig. 10. Typescript type system

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

Fig. 11. Typescript type convertibility

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

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

Fig. 12. Typescript translation

Used by languages such as TypeScript [CITEME] and Hack [CITEME], optional type systems provide easy backwards compatibility, with static checks being used where applicable. In our formalization, based on TypeScript's semantics, equivalence is structural and subtyping of recursive types is supported. The following three expressions illustrate the language. The first expression is ill-typed because method the receiver of the invocation to n is of type C and C does not have a method by that name. The second expression will also be flagged as erroneous as the argument to m is C rather than a D as expected. The last expression is statically correct as the instance of C is cast to a D via method m'. At runtime that expression will evaluate successfully.

$$
\begin{array}{ll}
\textbf{new } \mathsf{C}().\mathsf{n}(\textbf{new } \mathsf{D}()) & \textbf{where} \quad \mathsf{K} \;=\; \textbf{class } \mathsf{C} \; \{ \\
\textbf{new } \mathsf{C}().\mathsf{m}(\textbf{new } \mathsf{C}()) & \quad \mathsf{m}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{C} \; \{ \; \mathsf{this} \; \} \\
\textbf{new } \mathsf{C}().\mathsf{m}(\textbf{new } \mathsf{C}().\mathsf{m}'(\textbf{new } \mathsf{C}())) & \quad \mathsf{m}'(\mathsf{x}\colon \star)\colon \mathsf{D} \; \{ \; \mathsf{x} \; \} \\
 & \} \\
 & \textbf{class } \mathsf{D} \; \{ \; \}
\end{array}
$$

Our formalization of TypeScript's type system is in Fig. 10. The type system relies on the *convertibility* relation, denoted $\mathsf{K} \vdash_s \mathsf{t} \Rightarrow \mathsf{t}'$, which captures precisely the implicit type conversions allowed by TypeScript. The relation appears in Fig. 11 and states that a type is convertible to a super type and that $\star$ is convertible to anything and conversely. We illustrate the type system by focusing on the example above. For the first expression, the receiver has type C, but the class does not have method n. The second expression is ill-typed because the argument to method m is of type C and C is not convertible to D. The third expression is correct because the argument to m' is of type C and C is convertible to $\star$, the expected type of m'.

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.

The translation to KafKa appears in Fig. 12 and is straightforward. The translation function $[\![\ldots]\!]$ is applied to class definitions and expression, it takes TypeScript and returns the corresponding KafKa code. Grey background is used to identify translated terms. The translation is exceedingly simple, all type annotations become $\star$ and all source method calls are translated to dynamic KafKa calls. For instance one would translate the class definition and the last expression of the example above as:

$$
\begin{array}{ll}
(<\star> \textbf{new } \mathsf{C}())@\mathsf{m}_{\star\to\star}((<\star> \textbf{new } \mathsf{C}())@\mathsf{n}_{\star\to\star}(\textbf{new } \mathsf{C}())) & \textbf{where} \quad \mathsf{K} \;=\; \textbf{class } \mathsf{C} \; \{ \\
 & \quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star \; \{ \; <\star> \mathsf{this} \; \} \\
 & \quad \mathsf{m}'(\mathsf{x}\colon \star)\colon \star \; \{ \; \mathsf{x} \; \} \\
 & \}
\end{array}
$$

Observe that the TypeScript translations 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 TypeScript 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 TypeScript programs can get stuck at any point of their execution.

All TypeScript types are represented with the dynamic KafKa type, $\star$, and all translated expressions have type $\star$. This is a simple instance of a general property that all our translations into KafKa satisfy. Let $\mathsf{kty}(\mathsf{t})$ be a mapping from source types to KafKa types – for TypeScript the mapping is trivial: for all types t, $\mathsf{kty}(\mathsf{t}) = \star$. It then holds that if e is a well-typed source expression with type t, then $[\![\mathsf{e}]\!]$ is a well-typed KafKa expression with type $\mathsf{kty}(\mathsf{t})$.

5.2 Concrete

Concrete gradual type systems strictly enforce type membership. In a concrete gradual type system, only objects that arise from static subtypes of a type are allowed to inhabit that type. We will use Thorn as an example of a concretely gradually typed language.

Thorn has a combination of dynamic, optional, and concrete types. Concrete types, written C, behave as one would expect: a variable $\mathsf{x}\colon \mathsf{C}$ is guaranteed to refer to an instance of C or a subtype thereof. Optional types are analogous of TypeScript type annotations: occurrences of optionally typed variables, denoted $\mathsf{x}\colon ?\mathsf{C}$, are checked statically within their scope but may be bound to dynamic values. Optional types thus provide some of the benefits of static typing without any loss of expressiveness or flexibility. Subtyping on optional types is inherited from the subtyping relation on concrete types, that is $?\mathsf{C} <: ?\mathsf{D}$ whenever $\mathsf{C} <: \mathsf{D}$; also, it always hold that $\mathsf{C} <: ?\mathsf{C}$.

The formalization of the Thorn type system is built on top of the rules presented for TypeScript in Fig. 11 and Fig. 10. The definition of subtyping is extended to account for optional types, this appears in Fig. 13. 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. 14, this extra rule states that an optional type is convertible to a concrete parent type. Type rule for method calls, shown in Fig. 15 must be extended to handle receivers of optional types, they are treated as if they were concrete types.

```
new A().n(new D())
new A().m(new D())
new A().m(new A().n(new C()))
new A().n(new A().m(new C()))
```

```
where K = class A {
            n(x: C): C { x }
            m(x: ?C): ?C { x }
          }
          class C { m′(x: ⋆): ⋆ {x} }
          class D { m(x: ⋆): ⋆ {x} }
```

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

Fig. 13. Thorn subtyping

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

Fig. 14. Thorn type convertibility

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

Fig. 15. Thorn type system

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

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

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

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

$$\text{empty}\quad \textbf{otherwise}\ ..$$

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

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

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

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

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

$$\mathsf{t}_2 = \mathsf{kty}(\mathsf{t}_1)\quad \mathsf{t}'_2 = \mathsf{kty}(\mathsf{t}'_1)$$

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

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

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

Fig. 16. Thorn translation

The following expressions illustrate the difference between Thorn and TypeScript. The class table K has three classes, class A defines methods m and n which both return their argument. The only difference between them is that m uses optional types for its argument and return value, in both cases $?\mathsf{C}$. Classes C and D are defined such that they are unrelated by subtyping. We will study the four expressions, the first two are ill-typed, the remaining two are deemed well-typed, but the last one will require dynamic checks.

The first expression is ill-typed because method n expects a C, it gets a D which is not convertible to C. The second expression is ill-typed because it attempts to convert a D into a $?\mathsf{C}$. Typing the argument of m as a $?\mathsf{C}$ suggests that the intention of the programmer is to invoke this method only with instances that behave as C; since D does not convert to C the expression is flagged as erroneous. The third expression passes C where $?\mathsf{C}$ is expected; this is allowed as subtyping implies convertibility. The last expression invokes n and passes a $?\mathsf{C}$ where a C is expected; although potentially unsound, this is allowed by Thorn to foster program evolution. Convertibility thus includes a $?\mathsf{C} \Rightarrow \mathsf{C}$ rule; at runtime Thorn protects the call and ensures soundness by inserting a dynamic type cast to C around the argument. If the receiver object has an optional type, method invocation is statically checked: optional types behave as contracts between variables and contexts, and whenever an object has type $?\mathsf{C}$ a well-typed context is expected to use it only as a variable pointing to an instance of the class C. Since the runtime does not guarantee that $?\mathsf{C}$ variables points to instances of the class C, the conformance of the value actually accessed is checked individually at each method invocation, as if the callee had type $\star$.

The translation of Thorn into KafKa is given in Fig. 16. 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 $\mathsf{e.m}(\mathsf{e}')$. If the type of e is a concrete C, then a statically resolved invocation of the form $\mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$ will be emitted. If e is dynamic or an optional type, then a dynamically resolved call of the form $\mathsf{e@m}_{\star\to\star}(\mathsf{e}')$ 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.

We have seen that any method can be called statically and dynamically. For this to work out it is necessary to generate a dynamic version of all Thorn meth-

ods. Class translation relies on overloading to provide two implementations of each method: one is used with concrete receiver and the other is optionally typed (or dynamic) object. More precisely, given a Thorn method md with type $\mathsf{t}_1 \to \mathsf{t}_2$, the KafKa 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), to be used in a concretely typed context. The KafKa translation also generates a second method md'', that wraps md' so that it is safe to invoke it with type $\star \to \star$, that is from an optionally typed or dynamic context. Finally, each field $\mathsf{f}{:}\mathsf{t}$ is mapped to the corresponding $\mathsf{f}{:}\mathsf{kty}(\mathsf{t})$ field. As an example, the above translates to:

$$\begin{array}{l}\textbf{new } \mathsf{A}().\mathsf{n}_{\mathsf{C}\to\mathsf{C}}(<\mathsf{C}>(\textbf{new } \mathsf{A}().\mathsf{m}_{\star\to\star}(<\star>\textbf{new } \mathsf{C}()))) \\ \qquad\qquad \textbf{where}\quad \mathsf{K} = \textbf{class } \mathsf{A}\ \{ \\ \qquad\qquad\qquad \mathsf{m}(\mathsf{x}{:}\star){:}\star\ \{\ \mathsf{x}\ \} \\ \qquad\qquad\qquad \mathsf{n}(\mathsf{x}{:}\mathsf{C}){:}\mathsf{C}\ \{\ \mathsf{x}\ \} \\ \qquad\qquad\qquad \mathsf{n}(\mathsf{x}{:}\star){:}\star\ \{<\star>\mathsf{this}.\mathsf{n}_{\mathsf{C}\to\mathsf{C}}(<\mathsf{C}>\mathsf{x})\} \\ \qquad\qquad \}\end{array}$$

As expected, the outer invocation to n has been protected by a structural cast to C, illustrating how Thorn relies on overloading and the different invocation forms to protect its type invariants from untyped code.

5.3 Behavioral

The Behavioral semantics require that, to be of a type, a value must behave as if it actually was of that type, up to cast errors. We will consider Typed Racket, a language with behavioral semantics, as a representative example of languages that use this design.

$$\begin{array}{lll}
[\![\textbf{class } \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}_1..\,\}]\!] = \textbf{class } \mathsf{C}\,\{\,\mathsf{fd}_1..\ \mathsf{md}'_1..\,\} & & \\
\qquad\qquad \textbf{where}\quad \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}{:}\mathsf{t}){:}\mathsf{t}'\ \{\mathsf{e}'_1\}\ ..\quad \mathsf{md}_1 = \mathsf{m}(\mathsf{x}{:}\mathsf{t}){:}\mathsf{t}'\ \{\mathsf{e}_1\}\ .. & & \\
\qquad\qquad\quad \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}' & \textbf{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) & \textbf{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) & \textbf{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\quad \mathsf{m}(\mathsf{D}_1){:}\,\mathsf{D}_2 \in \mathsf{K}(\mathsf{C}) \\
[\![\textbf{new } \mathsf{C}(\mathsf{e}_1..)]\!]_\Gamma = \textbf{new } \mathsf{C}(\mathsf{e}'_1..) & \textbf{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}' & \textbf{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} & \textbf{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}$$

Fig. 17. Typed Racket translation

Typed Racket is an extension of the Racket programming language, supporting Racket's key functionality through a system of contracts that dynamically enforces type guarantees. At first brush the formalization may appear exceedingly simple. The Typed Racket static type system is identical to that of TypeScript. The translation, shown in Fig. 17 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 $(\!|\mathsf{e}|\!)^{\mathsf{t}}_{T}$ 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. To illustrate the semantics of Typed Racket, consider the following example:

$$\begin{array}{lll}\textbf{new } \mathsf{A}().\mathsf{m}(\textbf{new } \mathsf{C}()) & \textbf{where} \quad \mathsf{K} & = \textbf{class } \mathsf{A} \{ \\ & & \quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star \{ \textbf{ new } \mathsf{D}().\mathsf{m}(\mathsf{x}) \ \} \\ & & \} \\ & & \textbf{class } \mathsf{C} \{ \ \mathsf{m}(\mathsf{x}\colon \star)\colon \star \{ \ \mathsf{x} \ \} \} \\ & & \textbf{class } \mathsf{D} \{ \ \mathsf{m}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{D} \ \{ \ \mathsf{x}.\mathsf{m}(\mathsf{x}) \ \} \}\end{array}$$

Values cross several typed/untyped boundaries in this example. For instance, A's method m expects a value of type $\star$ while the method m of D expects a value of type D. The resulting translation will insert a behavioral cast from $\star$ to D at the call site. In order to protect class D from misbehaving objects, Typed Racket inserts a wrapper object around the argument of D; the wrapper is responsible for ensuring that the actual value respects the constraints of class D. The KafKa translation for Typed Racket inserts a behavioral cast at the boundary between the invocation of class D's method m in class A. This has the same effect of Typed Racket wrapping: the object with type C is wrapped to make sure that it acts as an object of type D. The generated KafKa code for the example is:

$$\begin{array}{lll}\textbf{new } \mathsf{A}().\mathsf{m}_{\star\to\star}(\blacktriangleleft \star \blacktriangleright \textbf{new } \mathsf{C}()) & & \\ \qquad\qquad \textbf{where} \quad \mathsf{K} & = & \textbf{class } \mathsf{A} \{ \\ & & \quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star \{ \ \blacktriangleleft \star \blacktriangleright \textbf{ new } \mathsf{D}().\mathsf{m}_{\mathsf{D}\to\mathsf{D}}(\blacktriangleleft \mathsf{D} \blacktriangleright \mathsf{x}) \ \} \\ & & \} \\ & & \textbf{class } \mathsf{D} \{ \ \mathsf{m}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{D} \ \{\mathsf{x}.\mathsf{m}_{\mathsf{D}\to\mathsf{D}}(\mathsf{x})\} \} \\ & & \textbf{class } \mathsf{C} \{ \ \mathsf{m}(\mathsf{x}\colon \star)\colon \star \ \{\mathsf{x}\} \}\end{array}$$

By enforcing types on the values and not leaving the check that the value has the correct types to the user, Typed Racket is able to translate the typed code in class D as if the language had no untyped code. Any misbehavior will be instantly caught by the value themselves.

As mentioned earlier, we depart from Typed Racket in several ways. In Type Racket the granularity of boundaries is the module, all classes in the same module are either typed or untyped. The finer granularity adopted here does not lose expressiveness and matches the approach adopted in the Guarded variant of Reticulated Python. Another feature of Typed Racket, that can be switched on or off is protection of the this reference. When that feature is on, accesses to the this reference are protected. This is needed to support inheritance – if an untyped class inherits from a typed one, then the self-reference may be accessed from both typed and untyped contexts without necessarily passing through a typed/untyped boundary. As a result, inheritance in a gradually typed setting requires the protection of all self-references.

5.4 Transient Python

The Transient variant of Reticulated Python aims for a form soundness with a predictable cost model. The intuition for the Transient guarantee is that, in a call such as $\mathsf{e.m(e')}$, if the type of e is some class C, then that method invocation will succeed. More precisely, if e evaluates to a value a, the object denoted by a has a method m. Of course, if the type of e is $\star$, then the call can get stuck on a method not found. Transient obtains that guarantee by statically checking that every expression is consistent to the expected type and dynamically checking that expressions evaluates to a value that has the method requested.

The Transient static type system is based on TypeScript except that the convertibility rules now builds on an auxiliary *consistency* relation, defined in Fig. 19 to relate types t and $\mathsf{t'}$. The modified convertibility rule appears in Fig. 18. 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.

The Transient translation appears in Fig. 20. 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 $\mathsf{e.m(e')}$ is translated to $\mathsf{e.m_{\star\to\star}(e')}$ 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 $\mathsf{e@m_{\star\to\star}(e')}$ 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 $<\mathsf{t}>\mathsf{x};\ \mathsf{e}$ where e is the body of the expression and the semi

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

Fig. 18. Transient convertibility

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

$$\frac{\begin{array}{c}\mathsf{M}' = \mathsf{M}\ \mathsf{C} <: \mathsf{D} \\ \mathsf{mt} \in \mathsf{K}(\mathsf{D}) \implies \mathsf{mt}' \in \mathsf{K}(\mathsf{C}) \quad \wedge \quad \mathsf{M}'\ \mathsf{K} \vdash \mathsf{mt} \lesssim \mathsf{mt}'\end{array}}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{C} \lesssim \mathsf{D}}$$

$$\frac{\mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_2 \lesssim \mathsf{t}_1 \qquad \mathsf{M}\ \mathsf{K} \vdash \mathsf{t}_1' \lesssim \mathsf{t}_2'}{\mathsf{M}\ \mathsf{K} \vdash \mathsf{m}(\mathsf{t}_1) : \mathsf{t}_1' \lesssim \mathsf{m}(\mathsf{t}_2) : \mathsf{t}_2'}$$

Fig. 19. Transient consistent subtyping

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

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

Fig. 20. Transient translation

colon is syntactic sugar for sequencing.[2] Likewise, in order to ensure that field

[2] This can be sugared to $<\mathsf{t}'>\ \textbf{new}\ \mathsf{A}(<\star><\mathsf{t}>\mathsf{x}).\mathsf{m}_{\star\to\star}(<\star>\mathsf{e})$ where A is class with a single field of type $\star$ and an identity method m. The type of e is t'. All of the cast expect the one to t are only there to please the type system and can be optimized away as they will always succeed.

assignment will not get stuck, Transient gives all fields type $\star$, then checks the field value on reads based on the static typing.

Source	**KafKa**
class C { m(x: C): C {x} }	**class** C { m(x: $\star$): $\star$ {<C> x; <C> x} }
class D { n(x: D): D {x} }	**class** D { n(x: $\star$): $\star$ {<D> x; <D> x} }
class E { m(x: D): D {x} }	**class** E { m(x: $\star$): $\star$ {<D> x; <D> x} }
class F { m(x: E): E {x}	**class** F { m(x: $\star$): $\star$ {<E> x; <E> x}
n(x: $\star$): $\star$ {this.m(x)} }	n(x: $\star$): $\star$ {<$\star$> x; <E> this.$\mathsf{m}_{\star\to\star}$(<$\star$> x)} }
new F().n(**new** C()).m(**new** C())	**new** F().$\mathsf{n}_{\star\to\star}$(<$\star$> **new** C())@$\mathsf{m}_{\star\to\star}$(**new** C())

The example presents two mutually incompatible classes C and D, along two potential consumers for C and D, C itself and E, and a conversion class F. F is used to acquire a reference to C as an E, which implies that the method m of C has type D to D, despite it actually having type C to C, even with the possibility to call method m through untyped code.

In Typed Racket, any call to method m of class F, encounters a wrapper that ensures that the argument and return type of m is always a D. Therefore, if the method m is called with a C, the program would get stuck at the cast to D. In Transient, the cast to E is a no-op, so when we call m with C, no extra cast to D is encountered. The Transient design allows method m, which expects an instance of class C as argument, to be called with a value of type $\star$ at any point. However, this forces m to check its arguments at every method invocation.

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. Type annotations are a source of information that can be used to generate efficient code, but without soundness this cannot be relied upon and becomes, at best, hints. Some guarantees become apparent in our translations – when the translation generates static method invocations, for Thorn, Racket and Python, these invocations can be optimized. This comes at the cost of casts and wrappers. The nature of these casts and their dynamic frequency will make a significant difference to end users. Thorn inserts them at boundaries of concrete types only. Racket inserts them whenever a value crosses either a typed or untyped boundary. Transient Python inserts casts after variable reads.

References

1. 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.
2. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. 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.
4. Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming (ECOOP)*, 2014. doi:10.1007/978-3-662-44202-9_11.
5. 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. 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.
7. 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.
8. Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 1993. doi:10.1145/165854.165893.
9. Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004. doi:10.1145/1167473.1167479.
10. 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.
11. The Facebook Hack Team. Hack, 2016. http://hacklang.org.
12. The Dart Team. Dart programming language specification, 2016. http://dartlang.org.
13. Robert Findler and Matthias Felleisen. Contracts for higher-order functions. In *International Conference on Functional Programming (ICFP)*, 2002. doi:10.1145/581478.581484.

14. 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.
15. 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.
16. Jeremy G Siek and Philip Wadler. Threesomes, with and without blame. In *ACM Sigplan Notices*, volume 45, pages 365–376. ACM, 2010.

A Full Definition

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

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

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

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

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

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

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

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

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

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

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

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

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

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

KT-VAR

$$\frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{x} : \mathsf{t}}$$

KT-SUB

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}' \quad \cdot\ \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}$$

KT-READ

$$\frac{\Gamma(\mathsf{this}) = \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{this.f} : \mathsf{t}}$$

KT-REFREAD

$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{..\} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a.f} : \mathsf{t}}$$

KT-WRITE

$$\frac{\Gamma(\mathsf{this}) = \mathsf{C} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) \quad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{this.f} = \mathsf{e} : \mathsf{t}}$$

KT-REFWRITE

$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{..\} \quad \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) \quad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a.f} = \mathsf{e} : \mathsf{t}}$$

KT-CALL

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

KT-DYNCALL

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

KT-NEW

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}_1 : \mathsf{t}_1.. \quad \mathbf{class}\ \mathsf{C}\,\{\,\mathsf{f}_1 : \mathsf{t}_1..\ \mathsf{md}_1..\,\} \in \mathsf{K}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathbf{new}\ \mathsf{C}(\mathsf{e}_1..) : \mathsf{C}}$$

KT-SUBCAST

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash <\mathsf{t}> \mathsf{e} : \mathsf{t}}$$

KT-BEHCAST

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash \blacktriangleleft\ \mathsf{t}\ \blacktriangleright\ \mathsf{e} : \mathsf{t}}$$

KT-REFTYPE

$$\frac{\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a'}_1..\}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}}$$

KT-REFANY

$$\frac{}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \star}$$

A.3 Dynamic function

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

DYNE

$$\frac{}{\mathsf{dyn}(\cdot) = \cdot}$$

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).

SGE

$$\frac{}{\mathsf{signature}(\cdot) = \cdot}$$

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 syntax and semantics

Well-formedness for Thorn

The well-formedness judgments for Thorn is similar to the well-formedness judgments of KafKa with the addition of well-formed optional types (?C). The turnstile ($\vdash_s$) of all source language judgment is characterized with s.

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

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

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

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

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

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

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

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

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

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

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

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

Well-formedness for Typed Racket, TypeScript, Transient

The well-formedness judgments for Typed Racket, TypeScript, and Transient is a subset of the well-formedness judgment for KafKa.

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

SWF-PROG

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

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

SWF-CLASS

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

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

SWF-TYMETH

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

SWF-DYMETH

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

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

SWF-FIELD

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

$\boxed{\mathsf{K} \vdash_s\ \mathsf{t}\ \checkmark}$ 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 Litmus tests

In this section, we present the examples of the litmus test, present in Fig. 2 of section 3, in the source code of the four gradual typing languages: Thorn, TypeScript, Typed Racket, and Transient.

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());
```

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())
```

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

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())
```