Gradual Types for Objects Redux

ANONYMOUS AUTHOR(S)

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 several gradual type systems by translation to KafKa and discuss the implications of the respective designs.

1 INTRODUCTION

"Because half the problem is seeing the problem"

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]. Ten years later, many systems support the gradual addition of types to untyped object-oriented programs.[1]

A decade later, faithful 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, the performance implications of their implementation strategies appear daunting.[2] Predictably, developers of industrial languages have chosen to compromise on soundness to avoid degrading performance.[3]

This paper explores the design space of gradual type systems for object-oriented languages. We aim to expose the forces that influenced the design of existing systems and discuss the implications of key decisions. While there are practical challenges that, in the end, may prevent adoption of some of the more ambitious type systems, there may be workable compromises that have not yet been fully explored. 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

[1]Languages which allow mixing typed and untyped with objects include C# [4], Dart [5], DRuby [6], Hack [7], Gradualtalk [8], Reticulated Python [9], Safe Typescript [10], StrongScript [11], Thorn [12], Typed Racket [13], TypeScript [14].
[2]Languages with *sound* gradual type systems, *i.e.* type systems that ensure to catch all violations of the typing discipline either statically or dynamically, experience orders of magnitude performance pathologies, e.g. 5x for Gradualtalk [15], 10x Reticulated Python [9], 22x Safe Typescript [10], and 121x Typed Racket [16]. These numbers merely indicate the existence of configurations that hurt performance. More study is needed to understand how often these pathologies occur in the wild.
[3]Dart, Typescript and Hack are good examples, they all offer unchecked modes so that production code is able to run without runtime checks.

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.[4]

The type systems considered in this work are idealizations of TypeScript [14], Thorn [12], Typed Racket [17], and the Transient variant of Reticulated Python [9]. We selected these four points in the design space because the languages had open source implementations and the designs were influential. TypeScript was chosen as a representative of unsound systems that work by erasure, such as Dart and Hack. Thorn combines dynamic and static types in the same way as C#, an approach 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. We focus on core features of each type system, namely their treatment of objects, and explain which errors are caught statically and which ones are caught dynamically. One, perhaps surprising, observation is that there is no agreement on what constitutes a "correct" program. We illustrate this in Section 5 with litmus tests that differentiate between the type systems based on which runtime errors are generated. It is worth emphasizing that the litmus tests are well-typed in all of the respective source languages and that, in an untyped implementation, they all run without error. Dynamic errors encountered in them when running under the typed implementations vary due to differences in the runtime checking strategies employed by the languages, rather than a fundamental bug in the tests. The translation to KafKa makes those differences explicit and comparable in the same framework.

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, and when dynamism is required. Our contribution is thus four translations, which we prove generate well-typed code, 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 also entails the translation of all the classes needed to evaluate it. Additionally, we provide a static type system to the four source languages, and define their dynamic semantics via translation. While soundness of KafKa terms is not really controversial, soundness of the source gradual type systems is more interesting. In KafKa, a well-typed program can only get stuck on a cast or at a dynamically resolved call. 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 well-typed method call can get stuck, as every type is effectively $\star$. Thorn has *concrete types*; a well-typed program will not get stuck at statically typed method calls. 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 the call to go through, but may get stuck if the called function returns a value of the wrong type.

The design of KafKa is a contribution of this work. 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 (denoted C), which give rise to homonymous types, and the dynamic type $\star$. KafKa subtyping is structural with recursion,

[4]Both the JVM and the CLR have runtime type tests, reflective invocation based on method names and runtime code generation. Dynamic resolution was added to C# in version 4.0 [14]. In Java, invokedynamic allows for custom method dispatch. The languages have similar class loading mechanisms.

using the class table K to resolve cycles.[5] The heart of any gradual type system implementation is the explicit casts that are inserted at type boundaries. A structural cast is built-in to KafKa: the subtype cast < t > a checks if the object referenced by a is a subtype of type t. To support the Typed Racket translation, KafKa is extended with a *behavioral* cast operation, which creates wrappers that enforce a specific type. This behavioral cast, ◀ t ▶ a, ensures that every method m invoked on a respects the type t.[6] Protecting a entails runtime generation of a new class, interposing on calls to the target object. We give KafKa an operational semantics and prove soundness of its type system. With the exception of behavioral casts the proof is straightforward.

Threats to validity. Our study has limitations, but we believe that these do not invalidate our conclusions. The source languages were simplified down to their essence and some features were excluded to allow comparison. For example, papers on Thorn and C# claim unboxed access to primitives and direct field reads are crucial for performance. KafKa does not model primitives, and fields are always accessed directly from the current object. Typed Racket supports class composition with first-class classes, while KafKa does not even have inheritance. TypeScript has generics; we have chosen to leave these out of the source language for simplicity of exposition. Subtyping in Thorn and C# is nominal, whereas in KafKa it is structural, as required for consistent subtyping in Transient. We believe these features could be accommodated at the cost of some complexity, and that their addition would not substantially change the outcome of our comparison. We leave one interesting system out of this study, namely, the Monotonic variant of Reticulated Python [9], because it requires in-place update of the class of objects, a feature not supported in commercial virtual machines. An earlier version of this paper proposed a translation that used an additional level of indirection to model Monotonic, but that was neither intellectually satisfactory nor elegant. This points to the fact that Monotonic does require some virtual machine support. Some gradual type systems support powerful error reporting features that can assign blame to the particular typed/untyped boundary crossing that led to an error. We do not study blame here, as it is not uniformly supported by all the languages under scrutiny and, we believe, it is somewhat orthogonal to matters at hand. Lastly, while we prove the type soundness of KafKa and that our translations generate well-typed code, we do not establish a formal correspondence with the semantics of the source languages. We gained confidence in our results by testing the implementations of the respective languages and validating that they agree with our translation.

2 BACKGROUND

"If you know the enemy and know yourself..."

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 [18], which led to Bracha's notion of pluggable types [19]. For him, types exist solely to catch errors at compile-time, never affecting the runtime behavior of programs. The rationale for this is that types are viewed as an add-on that can be turned off without affecting semantics. In the words of Richards *et al.* [11], 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 it prevents type annotations from introducing errors, and it follows that type annotations do not effect

[5]One subtle design choice was to privatize fields. An earlier version of this paper materialized fields in types by adding getter and setter methods, adding substantial verbosity for little benefit. This arose from KafKa's support for transparent wrappers, since, if fields are exposed, the wrappers must be able to interpose on field accesses. This feature contributed substantial complexity to KafKa.

[6]Neither the JVM or the CLR support behavioral casts natively, but the casts can be encoded by a combination of class generation and dynamic loading. To validate this claim, our implementation of KafKa in the CLR provides runtime support for behavioral generation.

performance. The optional type systems currently in wide use include Hack [7], TypeScript [14] and Dart [5].

On the functional side, the ancestry is dominated by the work of Felleisen and his students. The Typed Scheme [17] design that later became Typed Racket is influenced by the authors' earlier work on higher-order contracts [20]. 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. The Typed Racket approach to gradual typing is thus quite different from optional types. Whenever a value crosses a boundary between typed and untyped code, it is wrapped in a contract that monitors its behavior. This ensures that the type of mutable values remains consistent with their declared type and that functions respect their declared interface. When a value misbehaves, blame can be assigned to the boundary the value crossed. The granularity of typing is the module, thus a module is either entirely typed or entirely untyped. This means that a compilation unit only deals with uniform code (typed or untyped) and that closely coupled functions co-located in a module will not incur boundary crossing costs.

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 $t \sim t'$ 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 [9] attempts to find a compromise between soundness and efficiency. The language has three modes: the *guarded* mode behaves as Racket with contracts applied to values, but without the module granularity of Racket. The *transient* mode performs shallow checks on reads and returns, only validating if the value obtained has matching method names, while ignoring argument types. Lastly, the *monotonic* mode is fundamentally different from the other modes. 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 [8], C# 4.0 [4], Thorn [12], and StrongScript [11]. 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 (i.e. $\star$) to the C# language and adds dynamically resolved method invocation when the receiver of method call is of type $\star$. This means that C# has a dynamic sublanguage that allows developers to write unchecked code, working alongside a strongly typed sublanguage in which values are guaranteed to be of their declared type. The implementation of C# in the .Net framework replaces $\star$ by the type object and adds casts where needed. A dynamically resolved method call operation is supported as part of the reflective interface of .Net. 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.[7] 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.

[7]The translation strategy is surprisingly close to what we present later in the paper. The main difference is that the JVM does not have a type $\star$ so, like in C#, `Object` is used instead.

	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.

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; this is likely due to the JavaScript roots of that language. Anecdotal evidence suggests that structural subtyping is rarely needed [11]. StongScript extends TypeScript but changes 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.[8] 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 [9]. 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.

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 the ill-typed values. A fitting analogy are the stack traces printed by Java when a program terminates abruptly. Developers

[8]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.

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 another feature, namely precise exceptions, which does come at a price as it inhibits some compiler optimizations. Recording blame is likely costly, but no data exists on its real 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 can be viewed as a worst case scenario for concrete types – i.e. there is a shallow cast at almost every call. Finally, languages with generative casts tend to suffer prohibitive slow downs in pathological cases.

3 KAFKA: A CORE CALCULUS

“Aux chenilles du monde entier et aux papillons qu’elles renferment”

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*.

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

Fig. 2. KafKa Syntax.

Syntax. KafKa is an object calculus that satisfies the above design requirement. Its syntax is given in Fig. 2. Types consist of class names C, D and the dynamic type, written ★. Class definitions have a class name and (possibly empty) sequences of field and method definitions, **class** C { fd_1.. md_1.. }. Field definitions consist of a field and its type, f : t. Method definitions have (for simplicity) a single argument and an expression, denoted m(x: t): t {e}. KafKa supports a limited form of overloading, allowing both a typed implementation and an untyped implementation for each method. Fields are private to objects, and can be accessed only from the object’s scope; reading a field is denoted this.f and writing a field is denoted this.f = e. The calculus supports both statically and dynamically resolved method invocation. A statically resolved call, denoted $\mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$, is guaranteed to succeed as the type system ensures that the expression e will evaluate to an object of a class which has a method m(t) : t′. A dynamically resolved call, $\mathsf{e@m}_{\star\to\star}(\mathsf{e}')$, can get stuck if the object that e evaluates to does not have a method m(★): ★. 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*, <t> e, denotes the usual subtype cast that dynamically type-checks its argument. The latter, the *behavioral cast*, ◀ t ▶ 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 e K ✓, 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 m(x: C): D {e} and one untyped m(x: ★): ★ {e}. The static semantics of KafKa is mostly

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

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

Fig. 3. 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. 4. KafKa static semantics (excerpt)

standard; the complete set of rules is in Appendix. The subtype relation, $\mathsf{M}\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$, shown in Fig. 3, 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. 4. 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. 5. To resolve the this reference in field accesses, the syntax of expressions is extended at runtime with forms

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

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

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

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 ⋆ 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.

K	**new** $C(a_1..)$	σ	$\longrightarrow$	K	a'	σ'	**where**	a' fresh	$\sigma' = \sigma[a' \mapsto C\{a_1..\}]$
K	$a.f_i$	σ	$\longrightarrow$	K	a_i	σ	**where**	$\sigma(a) = C\{a_1..a_i..\}$	
K	$a.f_i = a'$	σ	$\longrightarrow$	K	a'	σ'	**where**	$\sigma(a) = C\{a_1..a_i..\}$	$\sigma' = \sigma[a \mapsto C\{a_1..a'..\}]$
K	$a.m_{t \to t'}(a')$	σ	$\longrightarrow$	K	e'	σ	**where**	$e' = [a/this\ a'/x]e$ $\sigma(a) = C\{a_1..\}$	$m(x\colon t)\colon t'\ \{e\} \in K(C)$
K	$a@m_{\star\to\star}(a')$	σ	$\longrightarrow$	K	e'	σ	**where**	$e' = [a/this\ a'/x]e$ $\sigma(a) = C\{a_1..\}$	$m(x\colon \star)\colon \star\ \{e\} \in K(C)$
K	$<\star>\ a$	σ	$\longrightarrow$	K	a	σ			
K	$<D>\ a$	σ	$\longrightarrow$	K	a	σ	**where**	$K \vdash C <: D$	$\sigma(a) = C\{a_1..\}$
K	◀ t ▶ a	σ	$\longrightarrow$	K'	a'	σ'	**where**	$K'\ a'\ \sigma' = \mathrm{bcast}(a, t, \sigma, K)$	
K	$E[e]$	σ	$\longrightarrow$	K'	$E[e']$	σ'	**where**	$K\ e\ \sigma \longrightarrow K'\ e'\ \sigma'$	

Fig. 5. KafKa dynamic semantics

Behavioral Casts. A cast ◀ t ▶ 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 class (the wrapper itself). Its dynamic semantics is reported in Fig. 6 and Fig. 7. When the target type is a class, say C′, the bcast function wraps a reference a with an instance of a freshly generated wrapper class D that abides by the interface of C′ or gets stuck. The function allocates the wrapper in the heap and updates the class table. The cast performs a check that the source type has at least all the method names requested by the target type. Also the cast is not defined for classes with overloaded methods (nodups 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 md'_1.. is forwarded to the corresponding method invocation in md_1, except that the arguments are protected by behavioral casts following the interface in

$\mathsf{md_1..}$ and the return type following the interface in $\mathsf{md'_1}$... Methods defined in the source type but missing from $\mathsf{md'_1..}$ are added to the wrapper class and simply redirected to corresponding method in the wrapped object. Wrapping an object so that it behaves as $\star$ is simpler, as the wrapper systematically protects the input type and casts back the return type to $\star$. Only methods existing in the source type are wrapped. The behavioral cast reduction rule satisfies a property instrumental to our design: a wrapper class generated for a target type D, is always a subtype of D. This will allow us to refer to both unwrapped and wrapped objects via the original, source language, types. The wrapper generation function will always produce a well-formed class for a target type D. The reason being that the wrapper class only contain methods from either the source or the target type. The wrapper class will also never contain any duplicates as it can only contain methods that exists in the source type. Furthermore, at most the wrapper class will contain every method of the source type.

3.1 Type soundness

Well-typed KafKa configurations satisfy a type-preservation theorem, and either reduce to values or are stuck on a dynamic method invocation or on one of the two casts provided by the language.

Theorem: Type Soundness. Given $\mathsf{K\ e\ \sigma\ \checkmark}$ and $\mathsf{\cdot\,\sigma\,K \vdash e : t}$, **then** either one of the following holds: e is already evaluated to value a, there is some e′ such that $\mathsf{K\ e\ \sigma \longrightarrow K'\ e'\ \sigma'}$ and $\mathsf{K'\ e'\ \sigma'\ \checkmark}$ and $\mathsf{\cdot\,\sigma'\,K' \vdash e' : t}$, **or** e is stuck on one of the three following error contexts $\mathsf{E[a@m_{\star\to\star}(a')]}$, $\mathsf{E[<t'>\ a]}$, or $\mathsf{E[\blacktriangleleft\ t'\ \blacktriangleright\ a]}$.

The wrapper class generated from behavioral cast will not contain any duplicate methods or fields. The nodups function takes a list of method definitions, and will return false if there exists any duplicate method names.

Lemma: No duplicates. Given $\mathsf{md_1.. \in K(C)}$ and $\mathsf{nodups(md_1..)}$ and $\mathsf{names(md'_1..) \subseteq names(md_1..)}$, and $\mathsf{k = W(C, md_1.., md'_1.., D, that)}$, **then** $\mathsf{nodups(md''_1..)}$, where $\mathsf{md''_1.. \in k}$.

The wrapper class generated from behavioral cast is always a subtype of the target type.

Lemma: Subtype preservation. Given $\mathsf{md_1.. \in K(C)}$ and $\mathsf{md'_1.. \in K(C')}$ and $\mathsf{names(md'_1..) \subseteq names(md_1..)}$, and $\mathsf{W(C, md_1.., md'_1.., D, that)}$, **then** $\mathsf{\emptyset\ K \vdash D <: C'..}$

3.2 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

$$\mathsf{bcast(a, C', \sigma, K) = K'\ a'\ \sigma'} \quad \textbf{where} \quad \left\{ \begin{array}{l} \mathsf{\sigma(a) = C\{a_1..\} \quad D, a', that\ fresh \quad md_1.. \in K(C) \quad md'_1.. \in K(C')} \\ \mathsf{names(md'_1..) \subseteq names(md_1..) \quad nodups(md_1..) \quad nodups(md'_1..)} \\ \mathsf{K' = K\,W(C, md_1.., md'_1.., D, that) \quad \sigma' = \sigma[a' \mapsto D\{a\}]} \end{array} \right.$$

$$\mathsf{bcast(a, \star, \sigma, K) = K'\ a'\ \sigma'} \quad \textbf{where} \quad \left\{ \begin{array}{l} \mathsf{\sigma(a) = C\{a_1..\} \quad md_1.. \in K(C) \quad D, a', that\ fresh \quad nodups(md_1..)} \\ \mathsf{K' = K\,W\star(C, md_1.., D, that) \quad \sigma' = \sigma[a' \mapsto D\{a\}]} \end{array} \right.$$

Fig. 6. Behavioral casts

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

Fig. 7. Wrapper class generation

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 $K \vdash t <: t'$ for some t and t′. C# already handles the case where t or t′ is ★, via its dynamic type, which has the same semantics as ★ in KafKa. If $t = C$ and $t' = D$, 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 that match to 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 [21], combined with the inherent restrictions on what programs can be reasonably written in KafKa.

4 TRANSLATING GRADUAL TYPE SYSTEMS

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

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 among them.

To avoid unnecessary clutter, we represent the source languages using the common syntax reported in Fig. 8. This defines a simple object calculus similar to KafKa, but without method overloading and, most importantly, cast operations. When modeling the Thorn type system, the additional type ?C is added to the type grammar to denote Thorn *like types*. For each language will we also give a source-level type system; we write $\vdash_s$ to emphasize that this is the source level typing (and not the KafKa level typing). As the various type system are mostly similar we will only present their differences assuming that all the omitted rules are identical. Given well-type source level code, each of the four translation produces well-typed KafKa code.

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

Fig. 8. Common syntax of source languages

The common source language does not have explicit casts but it allows to write programs such as

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

Here, the method m takes an argument of type ★ and returns the same value under type C. If the top-level expression passes an instance of D into m, the result is going to be of the wrong type. All the source language would allow this program but they differ in whether it is should give rise to an error.

4.1 TypeScript

Typescript is a backward compatible extension of JavaScript with classes and type annotations. Type 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.

```
new C().n(new D())                  where   K  =  class C {
new C().m(new C())                                  m(x: D): C { this }
new C().m(new C().m′(new C()))                      m′(x: ★): D { x }
                                                  }
                                                  class D { }
```

Our formalization of TypeScript's type system is in Fig. 9. The type system relies on the *convertibility* relation, denoted $K \vdash_s t \Rrightarrow t'$, which captures precisely the implicit type conversions allowed by TypeScript. The relation appears in Fig. 10 and states that a type is convertible to a super type and that ★ 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 ★, 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. 11 and is straightforward. The translation function $\llbracket \ldots \rrbracket$ 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:

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

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

$$\frac{\Gamma(x) = t}{\Gamma\ K \vdash_s x : t} \quad \frac{\begin{array}{c}\Gamma(\mathsf{this}) = C\\ f : t \in K(C)\end{array}}{\Gamma\ K \vdash_s \mathsf{this}.f : t} \quad \frac{\begin{array}{c}f : t \in K(\Gamma(\mathsf{this}))\\ \Gamma\ K \vdash_s e : t'\\ K \vdash_s t' \Mapsto t\end{array}}{\Gamma\ K \vdash_s \mathsf{this}.f = e : t} \quad \frac{\begin{array}{c}\Gamma\ K \vdash_s e : \star\\ \Gamma\ K \vdash_s e' : t\end{array}}{\Gamma\ K \vdash_s e.m(e') : \star} \quad \frac{\begin{array}{c}\Gamma\ K \vdash_s e : C\\ \Gamma\ K \vdash_s e' : t\\ K \vdash_s t \Mapsto t_1\\ m(t_1)\colon t_2 \in K(C)\end{array}}{\Gamma\ K \vdash_s e.m(e') : t_2} \quad \frac{\begin{array}{c}f_1 : t_1.. \in K(C)\\ \Gamma\ K \vdash_s e_1 : t'_1..\\ K \vdash_s t'_1 \Mapsto t_1..\end{array}}{\Gamma\ K \vdash_s \mathbf{new}\ C(e_1..) : C}$$

Fig. 9. TypeScript type system

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

Fig. 10. TypeScript type convertibility

$$\llbracket \mathbf{class}\ C\ \{\ fd_1..\ md_1..\ \}\rrbracket = \boxed{\mathbf{class}\ C\ \{\ fd'_1..\ md'_1..\ \}}\ \ \mathbf{where}\ \ \begin{array}{lll} fd'_1 = \boxed{f\colon\star}\ .. & fd_1 = f\colon t.. & md_1 = m(x\colon t_1)\colon t_2\ \{e\}\\ md'_1 = \boxed{m(x\colon\star)\colon\star\ \{e'\}}\ .. & & e' = \llbracket e\rrbracket \end{array}$$

$$\begin{array}{lcll}
\llbracket \mathsf{this}.f\rrbracket & = & \mathsf{this}.f & \\
\llbracket \mathsf{this}.f = e\rrbracket & = & \mathsf{this}.f = e' & \mathbf{where}\ e' = \llbracket e\rrbracket\\
\llbracket \mathsf{this}\rrbracket & = & <\star>\ \mathsf{this} & \\
\llbracket x\rrbracket & = & x & \\
\llbracket e_1.m(e_2)\rrbracket & = & e'_1@m_{\star\to\star}(e'_2) & \mathbf{where}\quad e'_1 = \llbracket e_1\rrbracket \quad e'_2 = \llbracket e_2\rrbracket\\
\llbracket \mathbf{new}\ C(e_1..)\rrbracket & = & <\star>\ \mathbf{new}\ C(e'_1..) & \mathbf{where}\quad e'_1 = \llbracket e_1\rrbracket\ ..
\end{array}$$

Fig. 11. TypeScript translation

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 kty(t) be a mapping from source types to KafKa types – for TypeScript the mapping is trivial: for all types t, kty(t) = $\star$. It then holds that if e is a well-typed source expression with type t, then $[\![e]\!]$ is a well-typed KafKa expression with type kty(t).

4.2 Thorn

Thorn has a combination of dynamic, optional, and concrete types. Concrete types, written C, behave as one would expect: a variable x: 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 x: ?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 ?C <: ?D whenever C <: D; also, it always hold that C <: ?C.

The formalization of the Thorn type system is built on top of the rules presented for TypeScript in Fig. 10 and Fig. 9. The definition of subtyping is extended to account for optional types, this appears in Fig. 12. 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. 13, this extra rule states that an optional type is convertible to a concrete parent type. Type rule for method calls, shown in Fig. 14 must be extended to handle receivers of optional types, they are treated as if they were concrete types.

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 ?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.

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

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 ?C. Typing the argument of m as a ?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 ?C is expected; this is allowed as subtyping implies convertibility. The last expression invokes n and passes a ?C where a C is expected; although potentially unsound, this is allowed by Thorn to foster program evolution. Convertibility thus includes a ?C $\Rrightarrow$ 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,

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

Fig. 12. Thorn subtyping

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

Fig. 13. Thorn type convertibility

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

Fig. 14. Thorn type system

$$\begin{array}{l}
[\![\textbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}_1 ..\ \mathsf{md}_1 ..\ \}]\!] = \textbf{class}\ \mathsf{C}\ \{\ \mathsf{fd}'_1 ..\ \mathsf{md}'_1 ..\ \mathsf{md}''_1 ..\ \}\quad \textbf{where}\\
\qquad\qquad \mathsf{fd}'_1 = \mathsf{f} \colon \mathsf{kty}(\mathsf{t})\ .. \qquad\qquad \mathsf{fd}_1 = \mathsf{f} : \mathsf{t} ..\\
\qquad\qquad \mathsf{md}'_1 = \mathsf{m}(\mathsf{x} \colon \mathsf{kty}(\mathsf{t}_1)) \colon \mathsf{kty}(\mathsf{t}_2)\ \{\mathsf{e}'\}\ .. \qquad \mathsf{md}_1 = \mathsf{m}(\mathsf{x} \colon \mathsf{t}_1) \colon \mathsf{t}_2\ \{\mathsf{e}\} .. \qquad \mathsf{e}' = (\!|\mathsf{e}|\!)^{\mathsf{t}_2}_{\mathsf{thisC}\ \mathsf{x t}_1}\ ..\\
\qquad\qquad \mathsf{md}''_1 = \mathsf{m}(\mathsf{x} \colon \star) \colon \star\ \{<\star>\ \mathsf{this.m}_{\mathsf{t}_1 \to \mathsf{t}_2}(<\mathsf{kty}(\mathsf{t}_1)>\ \mathsf{x})\} \qquad \textbf{if}\ \ \mathsf{kty}(\mathsf{t}_1) = \mathsf{D}\ \ \textbf{or}\ \ \mathsf{kty}(\mathsf{t}_2) = \mathsf{D}\\
\qquad\qquad\qquad \text{empty}\quad \textbf{otherwise}\ ..
\end{array}$$

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

Fig. 15. Thorn translation

and whenever an object has type ?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 ?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. 15. 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 $(\!|e|\!)^{t}_{\Gamma}$ 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 e.m(e′). If the type of e is a concrete C, then a statically resolved invocation of the form $e.m_{t \to t'}(e')$ will be emitted. If e is dynamic or an optional type, then a dynamically resolved call of the form $e@m_{\star\to\star}(e')$ 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 kty(t) function: kty(t) = ⋆ if t = ?C or t =⋆, t otherwise. Thorn optional and dynamic types are mapped to the ⋆ 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 methods. 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 $t_1 \to t_2$, the KafKa translation first generates a method md′ with type $kty(t_1) \to kty(t_2)$ (protecting the the return value with appropriate cast to $kty(t_2)$ 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 f:t is mapped to the corresponding f:kty(t) field. As an example, the above translates to:

```
new A().n_{C→C}(<C> (new A().m_{⋆→⋆}(<⋆> new C())))
                                        where  K =  class A {
                                                      m(x: ⋆): ⋆ { x }
                                                      n(x: C): C { x }
                                                      n(x: ⋆): ⋆ {<⋆> this.n_{C→C}(<C> x)}
                                                    }
```

As expected, the outer invocation to n has been protected by a structural cast to C. Remark that KafKa overloading and the combination of concretely typed and dynamically typed method invocation, together with structural casts, are instrumental to give semantics to the Thorn language.

4.3 Typed Racket

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. 16 maps classes to homonymous classes and types to types of the same name. The kty(t) function is the identity. Calls with a receiver of type ⋆ are translated to KafKa dynamic calls, and calls with a receiver of some type C are translated to statically typed calls. The auxiliary type-directed translation function $(\!|e|\!)^{t}_{\Gamma}$ introduces behavioral casts to ⋆ 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:

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

Values cross several typed/untyped boundaries in this example. For instance, A's method m expects a value of type ★ while the method m of D expects a value of type D. The resulting translation will insert a behavioral cast from ★ 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:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{new}\ \mathsf{A}().\mathsf{m}_{\star\to\star}(\blacktriangleleft \star \blacktriangleright\ \textbf{new}\ \mathsf{C}())} \\
 & \textbf{where}\ \ \mathsf{K}\ = & \textbf{class}\ \mathsf{A}\,\{ \\
 & & \quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright\ \textbf{new}\ \mathsf{D}().\mathsf{m}_{\mathsf{D}\to\mathsf{D}}(\blacktriangleleft \mathsf{D} \blacktriangleright\ \mathsf{x})\ \} \\
 & & \} \\
 & & \textbf{class}\ \mathsf{D}\,\{\ \mathsf{m}(\mathsf{x}\colon \mathsf{D})\colon \mathsf{D}\ \{\mathsf{x}.\mathsf{m}_{\mathsf{D}\to\mathsf{D}}(\mathsf{x})\}\ \} \\
 & & \textbf{class}\ \mathsf{C}\,\{\ \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{x}\}\ \}
\end{array}$$

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

$$\begin{array}{llllll}
\llbracket \textbf{class}\ \mathsf{C}\,\{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \}\rrbracket = \textbf{class}\ \mathsf{C}\,\{\ \mathsf{fd}_1..\ \mathsf{md}'_1..\ \} & & & & & \\
 & \textbf{where} & \mathsf{md}'_1 = \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\mathsf{e}'_1\}\ .. & & \mathsf{md}_1 = \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\mathsf{e}_1\}\ .. & \\
 & & \quad \mathsf{e}'_1 = \llbracket \mathsf{e}_1\rrbracket_{\mathsf{x}:\mathsf{t}\ \mathsf{this}:\mathsf{C}} & & & \\
\llbracket \mathsf{x}\rrbracket_\Gamma = \mathsf{x} & & & & & \\
\llbracket \mathsf{this}.\mathsf{f}\rrbracket_\Gamma = \mathsf{this}.\mathsf{f} & & & & & \\
\llbracket \mathsf{this}.\mathsf{f} = \mathsf{e}\rrbracket_\Gamma = \mathsf{this}.\mathsf{f} = \mathsf{e}' & \textbf{where} & \mathsf{K} \vdash \mathsf{this} : \mathsf{C} & \mathsf{e}' = \llparenthesis \mathsf{e}\rrparenthesis^{\mathsf{t}}_\Gamma & \mathsf{f} : \mathsf{t} \in \mathsf{K}(\mathsf{C}) & \\
\llbracket \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)\rrbracket_\Gamma = \mathsf{e}'_1@\mathsf{m}_{\star\to\star}(\mathsf{e}'_2) & \textbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e}_1 : \star & \mathsf{e}'_1 = \llbracket \mathsf{e}_1\rrbracket_\Gamma & \mathsf{e}'_2 = \llparenthesis \mathsf{e}_2\rrparenthesis^{\star}_\Gamma & \\
\llbracket \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)\rrbracket_\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} & \mathsf{e}'_1 = \llbracket \mathsf{e}_1\rrbracket_\Gamma & \mathsf{e}'_2 = \llparenthesis \mathsf{e}_2\rrparenthesis^{\mathsf{D}_1}_\Gamma & \mathsf{m}(\mathsf{D}_1)\colon \mathsf{D}_2 \in \mathsf{K}(\mathsf{C}) \\
\llbracket \textbf{new}\ \mathsf{C}(\mathsf{e}_1..)\rrbracket_\Gamma = \textbf{new}\ \mathsf{C}(\mathsf{e}'_1..) & \textbf{where} & \mathsf{e}'_1 = \llparenthesis \mathsf{e}_1\rrparenthesis^{\mathsf{t}_1}_\Gamma\ .. & \mathsf{f}_1 : \mathsf{t}_1 \in \mathsf{K}(\mathsf{C})\ .. & & \\
\llparenthesis \mathsf{e}\rrparenthesis^{\mathsf{t}}_\Gamma = \mathsf{e}' & \textbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' & \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}' & \mathsf{e}' = \llbracket \mathsf{e}\rrbracket_\Gamma & \\
\llparenthesis \mathsf{e}\rrparenthesis^{\mathsf{t}}_\Gamma = \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} & \textbf{where} & \mathsf{K},\Gamma \vdash \mathsf{e} : \mathsf{t}' & \mathsf{K} \vdash \mathsf{t} \not<: \mathsf{t}' & \mathsf{e}' = \llbracket \mathsf{e}\rrbracket_\Gamma &
\end{array}$$

Fig. 16. Typed Racket translation

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.

4.4 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 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. 18 to relate types t and t′. The modified convertibility rule appears in Fig. 17. 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. 19. Each class is translated to a homonymous KafKa class, field types are translated to $\star$, method types are given the $\star \to \star$ signature. Since argument and return types are erased, our translation can use structural casts to implement Transient runtime checks. They degenerate to simple inclusion checks on method names. The translation of method invocation makes explicit the Transient guarantee. A method call e.m(e′) is translated to e.m$_{\star\to\star}$(e′) if the type e is not $\star$. This translation combined with the soundness of KafKa entails that the method call will not get stuck. Of course of e is of type $\star$, the call will be translated to e@m$_{\star\to\star}$(e′) which can get stuck. To achieve that guarantee the translation must insert structural casts at every expression read. Transient also checks arguments of methods, since the expression may not use the argument (but Transient checks it anyway) our translation generates method bodies of the form < t > x; e where e is the body of the expression and the semi colon is syntactic sugar for sequencing.[9] Likewise, in order to ensure that field assignment will not get stuck, Transient gives all fields type $\star$, then checks the field value on reads based on the static typing.

```
                 Source                                        KafKa
    class C { m(x: C): C {x} }                class C { m(x: ★): ★ {<C> x; <C> x} }
    class D { n(x: D): D {x} }                class D { n(x: ★): ★ {<D> x; <D> x} }
    class E { m(x: D): D {x} }                class E { m(x: ★): ★ {<D> x; <D> x} }
    class F { m(x: E): E {x}                  class F { m(x: ★): ★ {<E> x; <E> x}
                n(x: ★): ★ {this.m(x)} }                  n(x: ★): ★ {<★> x; <E> this.m_{★→★}(<★> x)} }
    new F().n(new C()).m(new C())             new F().n_{★→★}(<★> new C())@m_{★→★}(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,

[9]This can be sugared to < t′ > **new** A(< $\star$ > < t > x).m$_{\star\to\star}$(< $\star$ > 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.

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

Fig. 17. Transient convertibility

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

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

Fig. 18. Transient consistent subtyping

$$\begin{array}{l}
[\![\mathbf{class}\ C\ \{\ fd_1..\ md_1..\ \}]\!] = \mathbf{class}\ C\ \{\ fd_1'..\ md_1'..\ \} \\
\qquad \mathbf{where}\quad fd_1' = f\colon \star\ ..\quad fd_1 = f\colon t..\quad md_1' = m(x\colon \star)\colon \star\ \{<t>\ x\ ;\ e_1'\}\ ..\quad md_1 = m(x\colon t)\colon t'\ \{e\}.. \\
\qquad\qquad e_1' = (\!|e|\!)^{t'}_{x:t\ this:C}\ ..
\end{array}$$

$$\begin{array}{llllllll}
[\![this]\!]_\Gamma & = & this & & & & & \\
[\![x]\!]_\Gamma & = & <t>\ x & \mathbf{where} & K,\Gamma \vdash x : t & & & \\
[\![this.f]\!]_\Gamma & = & <t>\ this.f & \mathbf{where} & K,\Gamma \vdash this : C & f\colon t \in K(C) & & \\
[\![this.f = e]\!]_\Gamma & = & <t>\ this.f = e' & \mathbf{where} & K,\Gamma \vdash this : C & f\colon t \in K(C) & e' = (\!|e|\!)^\star_\Gamma & \\
[\![e_1.m(e_2)]\!]_\Gamma & = & e_1'@m_{\star\to\star}(e_2') & \mathbf{where} & K,\Gamma \vdash e_1 : \star & e_1' = [\![e_1]\!]_\Gamma & e_2' = (\!|e_2|\!)^\star_\Gamma & \\
[\![e_1.m(e_2)]\!]_\Gamma & = & <t'>\ e_1'.m_{\star\to\star}(e_2') & \mathbf{where} & K,\Gamma \vdash e_1 : C & m(t)\colon t' \in K(C) & e_1' = [\![e_1]\!]_\Gamma & e_2' = (\!|e_2|\!)^\star_\Gamma \\
[\![\mathbf{new}\ C(e_1..)]\!]_\Gamma & = & \mathbf{new}\ C(e_1'..) & \mathbf{where} & f_1 : t_1 \in K(C) & e_1' = (\!|e_1|\!)^\star_\Gamma\ .. & & \\
(\!|e|\!)^t_\Gamma & = & <t>\ e' & \mathbf{where} & K,\Gamma \vdash e : t' & K \vdash t \lesssim t' & K \vdash t \not<: t' & e' = [\![e]\!]_\Gamma \\
(\!|e|\!)^t_\Gamma & = & e' & \mathbf{where} & K,\Gamma \vdash e : t' & K \vdash t <: t' & e' = [\![e]\!]_\Gamma &
\end{array}$$

Fig. 19. Transient translation

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 ⋆ at any point. However, this forces m to check its arguments at every method invocation.

5 GRADUAL TYPE SYSTEM SHOOTOUT & CONCLUSIONS

"There is no perfection only life."

Gradual typing has matured beyond being an experimental language feature in the petri dish of academia. Applications are now being written that incorporate gradual types. However the proposed language designs have subtle semantics and differ in their dark corners. It is the responsibility of language researchers to paint a clear picture of each approach to gradual typing.

	L1	L2	L3
Thorn			
Typed Racket		✓	
Transient RetPy		✓	✓
TypeScript	✓	✓	✓

L1

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

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

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

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

L2

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

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

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

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

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

L3

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

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

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

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

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

Fig. 20. Gradual typing semantic litmus tests.

In this paper we have compared the essence of four gradual type system designs, namely Typescript, Thorn, Typed Racket, and Transient Python. Our formalization have shown that they differ in which programs are viewed as statically correct. Thorn distinguishes between concrete and like types, whereas Transient uses the notion of consistency to determine if values may be compatible. A perhaps more surprising result is that even at runtime these languages do not agree on what constitutes a valid program. Consider Fig. 20 which shows three small programs – well-typed in all four languages. These synthetic examples consist of a class table and a top-level expression. All three will execute without getting stuck under TypeScript's permissive semantics. **L1** fails in all other semantics because A is not a subtype of I (Thorn) and A and I have different sets of methods (Racket and Python). **L2** fails in Thorn because the concrete type checks require subtyping of A and I to hold. **L3** fails in Thorn because n casts an instance of C to E, despite C not being a subtype of E. **L3** also fails in Racket as F.m casts an instance of C to E, which in turn requires C.m to only take and return values of type D, which is then called with an instance of a C. 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.

Another open question for gradual type system designers is performance of the resulting implementation. On the one hand, type annotations are a source of information about programmer intent that could be used to generate efficient code. On the other hand, without soundness guarantees this information cannot be relied upon and becomes, at best, hints that can be used in heuristic-driven code generation. The different languages do make some guarantees which become apparent in our translations – when the translation generate static method invocations, for Thorn, Racket and Python, these invocations can be optimized. The price for these guarantees comes in the form of casts and wrappers which are not be needed in a fully dynamic program. 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. Lastly, Transient Python inserts casts after variable reads. The casts themselves are different, Thorn and Python merely check for subtyping whereas Racket's casts require allocation.

This paper has introduced KafKa, a framework for comparing the designs of gradual type systems for object-oriented languages. We have proven KafKa's type system to be sound in the presence of the dynamic generation of wrapper classes. Then we have introduced four translations from idealized fragments of real programming language into KafKa. These translations are shown to generate well-typed KafKa code. The features that are required to support gradual types include subtype casts, the ability to generate new classes and for classes to transparently interpose on target objects. KafKa has proven to be a versatile vehicle for deconstructing the designs of gradual type systems.

Going forward there are some issues we wish to investigate further. We do not envision that supporting nominal subtyping within KafKa will pose problems, it would only take adding a nominal cast and changing the definition of classes. Then nominal and structural could coexist. A more challenging question is 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.

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

[5] The Dart Team. Dart programming language specification, 2016. http://dartlang.org.

[6] Michael Furr, Jong-hoon An, and Jeffrey Foster. Profile-guided static typing for dynamic scripting languages. In *Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2009. doi:10.1145/1640089.1640110.

[7] The Facebook Hack Team. Hack, 2016. http://hacklang.org.

[8] 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.

[9] 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.

[10] Aseem Rastogi, Nikhil Swamy, Cedric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe and efficient gradual typing for TypeScript. In *Symposium on Principles of Programming Languages (POPL)*, 2015. doi:10.1145/2676726.2676971.

[11] 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.

[12] 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.

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

[14] 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.

[15] Esteban Allende, Johan Fabry, and Éric Tanter. Cast insertion strategies for gradually-typed objects. In *Symposium on Dynamic languages (DLS)*, 2013. doi:10.1145/2508168.2508171.

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

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

[18] 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.

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

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

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

A FULL KAFKA DEFINITION

A.1 Well-formedness

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

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

$$\textsc{wp}\quad \frac{\begin{array}{c}\emptyset\,\sigma\,K \vdash e : t \\ K \vdash \sigma \checkmark \\ k \in K \implies \cdot\,K \vdash k \checkmark\end{array}}{K\ e\ \sigma \checkmark}$$

$\boxed{\sigma\ K \vdash \textbf{class}\ C\ \{\ fd_1..\ md_1..\ \}\ \checkmark}$ Well-formed c

$$\textsc{wc}\quad \frac{\begin{array}{c}K \vdash fd_1..\checkmark \\ this : C\ \ \sigma\,K \vdash md_1..\checkmark \\ nodups(md_1..\ fd_1..)\end{array}}{\sigma\,K \vdash \textbf{class}\ C\ \{\ fd_1..\ md_1..\ \}\ \checkmark}$$

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

$$\textsc{wt1}\quad \frac{\begin{array}{c}\Gamma' = \Gamma\ x{:}\,\star \\ \Gamma'\,\sigma\,K \vdash e : \star \qquad K \vdash \star \checkmark\end{array}}{\Gamma\,\sigma\,K \vdash m(x{:}\,\star){:}\,\star\ \{e\}\ \checkmark}$$

$$\textsc{wt2}\quad \frac{\Gamma' = \Gamma\ x{:}\,C \qquad \Gamma'\,\sigma\,K \vdash e : C' \qquad K \vdash C \checkmark \qquad K \vdash C' \checkmark}{\Gamma\,\sigma\,K \vdash m(x{:}\,C){:}\,C'\ \{e\}\ \checkmark}$$

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

$$\textsc{wf}\quad \frac{K \vdash t \checkmark}{K \vdash f{:}\,t \checkmark}$$

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

$$\textsc{wa}\quad \frac{}{K \vdash \star \checkmark} \qquad \textsc{wc}\quad \frac{C \in K}{K \vdash C \checkmark}$$

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

$$\textsc{wh}\quad \frac{\begin{array}{c}a' \mapsto C\{a_1..\}\ \in\ \sigma\ \implies \\ \textbf{class}\ C\ \{\ fd_1..\ md_1..\ \} \in K\ \wedge\ \cdot\,\sigma\,K \vdash a_1\end{array}}{K \vdash \sigma \checkmark}$$

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.

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

$$\text{KT-VAR}\ \frac{\Gamma(x) = t}{\Gamma\,\sigma\,K \vdash x : t} \qquad \text{KT-SUB}\ \frac{\begin{array}{c}\Gamma\,\sigma\,K \vdash e : t' \\ \cdot\ K \vdash t' <: t\end{array}}{\Gamma\,\sigma\,K \vdash e : t} \qquad \text{KT-READ}\ \frac{\begin{array}{c}\Gamma(\mathsf{this}) = C \\ f : t \in K(C)\end{array}}{\Gamma\,\sigma\,K \vdash \mathsf{this}.f : t}$$

$$\text{KT-REFREAD}\ \frac{\begin{array}{c}\sigma(a) = C\{..\} \\ f : t \in K(C)\end{array}}{\Gamma\,\sigma\,K \vdash a.f : t} \qquad \text{KT-WRITE}\ \frac{\begin{array}{c}\Gamma(\mathsf{this}) = C \\ f : t \in K(C) \\ \Gamma\,\sigma\,K \vdash e : t\end{array}}{\Gamma\,\sigma\,K \vdash \mathsf{this}.f = e : t}$$

$$\text{KT-REFWRITE}\ \frac{\begin{array}{c}\sigma(a) = C\{..\} \\ f : t \in K(C) \\ \Gamma\,\sigma\,K \vdash e : t\end{array}}{\Gamma\,\sigma\,K \vdash a.f = e : t} \qquad \text{KT-CALL}\ \frac{\begin{array}{c}\Gamma\,\sigma\,K \vdash e : C \\ \Gamma\,\sigma\,K \vdash e' : t \qquad m(t) : t' \in K(C)\end{array}}{\Gamma\,\sigma\,K \vdash e.m_{t \to t'}(e') : t'}$$

$$\text{KT-DYNCALL}\ \frac{\begin{array}{c}\Gamma\,\sigma\,K \vdash e : \star \\ \Gamma\,\sigma\,K \vdash e' : \star\end{array}}{\Gamma\,\sigma\,K \vdash e@m_{\star \to \star}(e') : \star}$$

$$\text{KT-NEW}\ \frac{\begin{array}{c}\Gamma\,\sigma\,K \vdash e_1 : t_1.. \\ \textbf{class}\ C\ \{\, f_1 : t_1..\ md_1..\,\} \in K\end{array}}{\Gamma\,\sigma\,K \vdash \textbf{new}\ C(e_1..) : C} \qquad \text{KT-SUBCAST}\ \frac{\Gamma\,\sigma\,K \vdash e : t'}{\Gamma\,\sigma\,K \vdash <t> e : t}$$

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

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

A.3 Dynamic function

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

$$\text{DYNE}\ \frac{}{\mathsf{dyn}(\cdot) = \cdot} \qquad \text{DYN}\ \frac{\mathsf{dyn}(mt_1..) = mt'_1..}{\mathsf{dyn}(m(t) : t\ \ mt_1..) = m(\star) : \star\ \ mt'_1..}$$

A.4 Signature function

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

$$\text{SGE}\ \frac{}{\mathsf{signature}(\cdot) = \cdot} \qquad \text{SG}\ \frac{md = m(x : t) : t\ \{e\} \qquad \mathsf{signature}(md_1..) = mt_1..}{\mathsf{signature}(md\ md_1..) = m(t) : t\ mt_1..}$$

A.5 Names function

The names function (names(fd_1..), names(md_1..), names(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 (nodups(mt_1..), nodups(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.

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

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

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

$$\textsc{wcl}\quad \frac{\begin{array}{c}\mathsf{nodups}(\mathsf{fd}_1, \mathsf{md}_1 ..) \qquad \mathsf{fd} \in \mathsf{fd}_1 .. \implies \mathsf{K} \vdash_s\ \mathsf{f} \\ \mathsf{md} \in \mathsf{md}_1 .. \implies \mathsf{this} : \mathsf{C}\ \mathsf{K} \vdash_s \mathsf{md}\,\checkmark\end{array}}{\mathsf{K} \vdash_s \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1 ..\ \mathsf{md}_1 ..\,\}\,\checkmark}$$

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

$$\textsc{wt}\quad \frac{\begin{array}{c}\Gamma\,\mathsf{x} : \mathsf{C}\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{D} \\ \mathsf{K} \vdash_s \mathsf{C}\,\checkmark \qquad \mathsf{K} \vdash_s \mathsf{D}\,\checkmark\end{array}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{m}(\mathsf{x} : \mathsf{C}) : \mathsf{D}\ \{\mathsf{e}\}\,\checkmark} \qquad \textsc{wwt}\quad \frac{\begin{array}{c}\Gamma\,\mathsf{x} : \mathsf{t}\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t}' \qquad \mathsf{K} \vdash_s \mathsf{t}\,\checkmark \\ \mathsf{K} \vdash_s \mathsf{t}'\,\checkmark \qquad \mathsf{kty}(\mathsf{t}) = \mathsf{kty}(\mathsf{t}') = \star\end{array}}{\Gamma\,\mathsf{K} \vdash_s \mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}'\ \{\mathsf{e}\}\,\checkmark}$$

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

$$\textsc{wf}\quad \frac{\mathsf{K} \vdash_s \mathsf{t}\,\checkmark}{\mathsf{K} \vdash_s \mathsf{f} : \mathsf{t}\,\checkmark} \qquad \textsc{wa}\quad \frac{}{\mathsf{K} \vdash_s \star\,\checkmark} \qquad \textsc{wc}\quad \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_s \mathsf{C}\,\checkmark} \qquad \textsc{ww}\quad \frac{\mathsf{C} \in}{\mathsf{K} \vdash_s\ ?}$$

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.

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

$$\textsc{swf-prog}\quad \frac{\begin{array}{c}\Gamma\ \cdot\ \mathsf{K} \vdash_s \mathsf{e} : \mathsf{t} \\ \mathsf{k} \in \mathsf{K} \implies \cdot\,\mathsf{K} \vdash_s \mathsf{k}\,\checkmark\end{array}}{\mathsf{e}\ \mathsf{K}\ \checkmark_s} \qquad \textsc{swf-class}\quad \frac{\begin{array}{c}\mathsf{nodups}(\mathsf{fd}_1 .., \mathsf{md}_1 ..) \qquad \mathsf{fd} \in \mathsf{fd}_1 .. \implies \mathsf{K} \vdash_s \mathsf{fd}\,\checkmark \\ \mathsf{md} \in \mathsf{md}_1 .. \implies \mathsf{this} : \mathsf{C}\ \ \mathsf{K} \vdash_s \mathsf{md}\,\checkmark\end{array}}{\mathsf{K} \vdash_s \textbf{class}\ \mathsf{C}\,\{\,\mathsf{fd}_1 ..\ \mathsf{md}_1 ..\,\}\,\checkmark}$$

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

$$\textsc{swf-tymeth}\quad \frac{\begin{array}{c}\Gamma\,\mathsf{x} : \mathsf{C}\ \ \mathsf{K}\,\mathsf{e} \vdash_s \mathsf{D} : \\ \mathsf{K} \vdash_s \mathsf{C}\,\checkmark \qquad \mathsf{K} \vdash_s \mathsf{D}\,\checkmark\end{array}}{\Gamma\ \mathsf{K} \vdash_s \mathsf{m}(\mathsf{x} : \mathsf{C}) : \mathsf{D}\ \{\mathsf{e}\}\,\checkmark} \qquad \textsc{swf-dymeth}\quad \frac{\begin{array}{c}\Gamma\,\mathsf{x} : \star\ \ \mathsf{K}\,\mathsf{e} \vdash_s \star : \\ \mathsf{K} \vdash_s \star\,\checkmark \qquad \mathsf{K} \vdash_s \star\,\checkmark\end{array}}{\Gamma\ \mathsf{K} \vdash_s \mathsf{m}(\mathsf{x} : \star) : \star\ \{\mathsf{e}\}\,\checkmark}$$

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

$$\frac{K \vdash_s t \checkmark}{K \vdash_s f : t \checkmark}\ \text{SWF-FIELD}$$

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

$$\frac{}{K \vdash_s \star \checkmark}\ \text{SWT-ANY} \qquad \frac{C \in K}{K \vdash_s C \checkmark}\ \text{SWT-TYPE}$$

C LITMUS TESTS IN SOURCE LANGUAGE

Thorn

Below we present the three litmus tests, presented in Fig. 20, in Thorn syntax.

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

Below we present the three litmus tests, presented in Fig. 20, in TypeScript syntax.

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

Below we present the three litmus tests, presented in Fig. 20, in Typed Racket syntax.

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

Below we present the three litmus tests, presented in Fig. 20, in Reticulated Python (transient) syntax.

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())
```

D PROOFS OF RELATED THEOREMS

In this section, we present the proof of type soundness for the type system of KafKa. The KafKa language includes the behavioural cast semantics shown in Fig. 6.

Theorem D.1 : Type Soundness of Core KafKa Typing

If:

a. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$

b. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{e} : \mathsf{t}$

then:

$\mathsf{K}\ \mathsf{e}\ \sigma \longrightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$

$\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$

$\cdot\ \sigma'\ \mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$

or:

$\mathsf{e} = \mathsf{a}$

or:

$\mathsf{e} = \mathsf{E}[\mathsf{a@m}_{\star\to\star}(\mathsf{a}')]$

or:

$\mathsf{e} = \mathsf{E}[<\mathsf{t}'>\ \mathsf{a}]$

or:

$\mathsf{e} = \mathsf{E}[\blacktriangleleft\ \mathsf{t}'\ \blacktriangleright\ \mathsf{a}]$

Theorem D.2 : Wrap function creates well-formed classes

If:

a. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$

b. $\mathsf{mt}_1.. = \mathsf{K}(\mathsf{C})$

c. $\mathsf{mt}'_1.. = \mathsf{K}(\mathsf{C}')$

d. $\mathsf{k} = \mathsf{W}(\mathsf{C}, \mathsf{mt}_1.., \mathsf{mt}'_1.., \mathsf{D}, \mathsf{that})$

e. $\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1..\}$

f. $\sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}]$

g. $\mathsf{nodups}(\mathsf{mt}_1..)$

h. that fresh

then:

$\sigma'\ \mathsf{K}\ \mathsf{k} \vdash \mathsf{k}\ \checkmark$

Lemma 1 : Extending Nodup with wrappers

If:

a. $\mathsf{k} = \mathsf{W}(\mathsf{C}, \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}', \mathsf{K}), \mathsf{D}, \mathsf{that})$

b. $\mathsf{nodups}(\mathsf{mtypes}(\mathsf{C}, \mathsf{K}))$

then:

$\mathsf{nodups}(\mathsf{mtypes}(\mathsf{D}, \mathsf{K}\ \mathsf{k}))$

Lemma 2 : Nodup of wrapper class gives overloading

If:

a. $\mathsf{k} = \mathsf{W}(\mathsf{C}, \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}', \mathsf{K}), \mathsf{D}, \mathsf{that})$

b. $\textbf{class}\ \mathsf{D}\ \{\ \mathsf{fd}_1..\ \mathsf{md}_1..\ \} \in \mathsf{K}\ \mathsf{k}$

c. $\mathsf{mt}_1.. = \mathsf{mtypes}(\mathsf{D}, \mathsf{K}\ \mathsf{k})$

d. $\text{nodups}(\text{mt}_1..)$
e. that fresh

then:

$\text{overloading}_{\emptyset}(\text{md}_1..\ \text{fd}_1..)$

Type Soundness of TypeScript and Typed Racket Translation

Lemma 3 : Type correctness of TypeScript/Typed Racket translation

If:

a. $K\ e \checkmark_s$
b. $K \vdash_s t \checkmark$
c. $x : t' \in \Gamma \implies K \vdash_s t' \checkmark$
d. $\Gamma\ K \vdash_s e : t$
e. $[\![K]\!] = K'$
f. $[\![\Gamma]\!] = \Gamma'$

then:

$\Gamma' \cdot K' \vdash [\![e]\!]_{\Gamma} : \star$

Lemma 4 : Soundness of TypeScript/Typed Racket convertibility

If:

a. $K \vdash_s t \Rrightarrow t'$
b. $\Gamma\ K \vdash_s e : t$
c. $e' = [\![e]\!]_{\Gamma}$
d. $[\![K]\!] = K'$
e. $[\![\Gamma]\!] = \Gamma'$
f. $\Gamma' \cdot K' \vdash e' : t$

then:

$\Gamma' \cdot K' \vdash (\!|e|\!)^{t'}_{\Gamma} : t'$

Lemma 5 : Type correctness of Typed Racket translation

If:

a. $K\ e \checkmark_s$
b. $K \vdash_s t \checkmark$
c. $x : t' \in \Gamma \implies K \vdash_s t' \checkmark$
d. $\Gamma\ K \vdash_s e : t$
e. $[\![K]\!] = K'$
f. $[\![\Gamma]\!] = \Gamma'$

then:

$\Gamma' \cdot K' \vdash [\![e]\!]_{\Gamma} : t$

Type Soundness of Thorn Translation

Lemma 6 : Soundness of convertibility

If:

a. $K \vdash t \Rrightarrow t'$
b. $\Gamma\ K \vdash_s e : t$
c. $e' = [\![e]\!]_{\Gamma}$

d. $[\![K]\!] = K'$
e. $[\![\Gamma]\!] = \Gamma'$
f. $\Gamma' \cdot K' \vdash e' : \mathrm{kty}(t)$

then:
$\Gamma' \cdot K' \vdash (\!|e|\!)_{\Gamma}^{t'} : \mathrm{kty}(t')$

Lemma 7 : Type correctness of Thorn translation

If:
a. $K\ e \checkmark_s$
b. $K \vdash_s t \checkmark$
c. $x : t' \in \Gamma \implies K \vdash_s t' \checkmark$
d. $\Gamma\ K \vdash_s e : t$
e. $[\![K]\!] = K'$
f. $[\![\Gamma]\!] = \Gamma'$

then:
$\Gamma' \cdot K' \vdash [\![e]\!]_{\Gamma} : \mathrm{kty}(t)$

Type Soundness of Transient Translation

Lemma 8 : Soundness of Transient convertibility

If:
a. $K \vdash_s t \Rrightarrow t'$
b. $\Gamma\ K \vdash_s e : t$
c. $e' = [\![e]\!]_{\Gamma}$
d. $[\![K]\!] = K'$
e. $[\![\Gamma]\!] = \Gamma'$
f. $\Gamma' \cdot K' \vdash e' : t$

then:
$\Gamma' \cdot K' \vdash (\!|e|\!)_{\Gamma}^{t'} : \mathrm{kty}(t')$

Lemma 9 : Type correctness of Transient translation

If:
a. $K\ e \checkmark_{tr}$
b. $K \vdash_s t \checkmark$
c. $x : t' \in \Gamma \implies K \vdash_s t' \checkmark$
d. $\Gamma\ K \vdash_s e : t$
e. $[\![K]\!] = K'$
f. $[\![\Gamma]\!] = \Gamma'$

then:
$\Gamma' \cdot K' \vdash [\![e]\!]_{\Gamma} : t$

Accessory Lemmas

Lemma 10 : Evaluation Extends Class Tables

If:
a. $K\ e\ \sigma \longrightarrow K'\ e'\ \sigma'$

then:

$K' = K\ K''$ for some K''

Lemma 11 : Weakening of Expression Typing over gamma

If:

a. $\Gamma\ \sigma\ K \vdash e : t$

then:

$\Gamma\ \Gamma'\ \sigma\ K \vdash e : t$

Lemma 12 : Weakening of Expression Typing over sigma

If:

a. $\Gamma\ \sigma\ K \vdash e : t$

then:

$\Gamma\ \sigma\ \sigma'\ K \vdash e : t$

Lemma 13 : Weakening of Expression Typing over class table

If:

a. $\Gamma\ \sigma\ K \vdash e : t$

then:

$\Gamma\ \sigma\ K\ K' \vdash e : t$

Lemma 14 : Weakening of Well-formedness for types over class table

If:

a. $K \vdash t\ \checkmark$

then:

$K\ K' \vdash t\ \checkmark$

Lemma 15 : Weakening of Well-formedness for methods over gamma

If:

a. $\Gamma\ \sigma\ K \vdash md\ \checkmark$

then:

$\Gamma\ \Gamma'\ \sigma\ K \vdash md\ \checkmark$

Lemma 16 : Weakening of Well-formedness for methods over sigma

If:

a. $\Gamma\ \sigma\ K \vdash md\ \checkmark$

then:

$\Gamma\ \sigma\ \sigma'\ K \vdash md\ \checkmark$

Lemma 17 : Weakening of Well-formedness for methods over class table

If:

a. $\Gamma\ \sigma\ K \vdash md\ \checkmark$

then:

$\Gamma\ \sigma\ K\ K' \vdash md\ \checkmark$

Lemma 18 : Weakening of Well-formedness for fields over class table

If:

a. $K \vdash f : t\ \checkmark$

then:

$\mathsf{K}\ \mathsf{K}' \vdash \mathsf{f} : \mathsf{t}\ \checkmark$

Lemma 19 : Weakening of Subtyping over class table

If:

a. $\mathsf{M}\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$

then:

$\mathsf{M}\ \mathsf{K}\ \mathsf{K}' \vdash \mathsf{t} <: \mathsf{t}'$

Lemma 20 : Weakening of Subtyping over methods

If:

a. $\mathsf{M}\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$

then:

$\mathsf{M}\ \mathsf{M}'\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$

Lemma 21 : Weakening of mtypes

If:

a. $\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}' \in \mathsf{K}(\mathsf{C})$

then:

$\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}' \in \mathsf{K}\ \mathsf{K}'(\mathsf{C})$ for some K'

Lemma 22 : Substitution

If:

a. $\mathsf{this} : \mathsf{C}\ \mathsf{x} : \mathsf{t}'\ \sigma\ \mathsf{K} \vdash \mathsf{e} : \mathsf{t}$

b. $\sigma(\mathsf{a}) = \mathsf{C}\{..\}$

c. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{a}' : \mathsf{t}'$

then:

$\cdot\ \sigma\ \mathsf{K} \vdash [\mathsf{a}/\mathsf{this}\ \mathsf{a}/\mathsf{x}]\mathsf{e} : \mathsf{t}$

Lemma 23 : Correctness of K(C)

If:

a. $\mathsf{m}(\mathsf{t}) : \mathsf{t}' \in \mathsf{K}(\mathsf{C})$

b. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{a} : \mathsf{C}$

c. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{a}' : \mathsf{t}$

d. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$

then:

$\mathsf{K}\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{a}')\ \sigma \longrightarrow \mathsf{K}\ \mathsf{e}''\ \sigma$

Lemma 24 : Canonical forms

If:

a. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$

b. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{a} : \mathsf{C}$

then:

$\sigma[\mathsf{a} \mapsto \mathsf{C}\{\underline{\mathsf{a}}\ \}]$

Lemma 25 : Evaluation retains typing

If:

a. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{e} : \mathsf{t}$
b. $\cdot\ \sigma\ \mathsf{K} \vdash \mathsf{e}' : \mathsf{t}'$
c. $\mathsf{K}\ \mathsf{e}'\ \sigma\ \checkmark$
d. $\mathsf{K}\ \mathsf{e}'\ \sigma \longrightarrow \mathsf{K}'\ \mathsf{e}''\ \sigma'$

then:

$\cdot\ \sigma'\ \mathsf{K}' \vdash \mathsf{e} : \mathsf{t}$, where $\mathsf{K}' = \mathsf{K}\ \mathsf{K}''$

Lemma 26 : Reduction preserves Well-formedness

If:

a. $\mathsf{K}\ \mathsf{e}\ \sigma \longrightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$
b. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$

then:

$\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$

Lemma 27 : Consistent Class Table

If:

a. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$

then:

$\forall$ **class** $\mathsf{C}\ \{\ \mathsf{fd}_1 ..\ \mathsf{md}_1 ..\ \}\ .\ \mathsf{K} \vdash \mathsf{fd}_1 ..\ \checkmark\ \wedge\ \cdot\ \sigma\ \mathsf{K} \vdash \mathsf{md}_1 ..\ \checkmark$