Gradual Types for Objects Redux

Anony Mouse, et al.

Abstract

The enduring popularity of dynamically typed languages has given rise to a cottage industry of static type systems, *gradual type systems*, that let developers annotate legacy code piecemeal. Soundness for a program which mixes typed and untyped code 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 serious challenges to implementers. 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.

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.00

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 differ in dynamic type annotations. Soundness at the boundaries between typed and untyped code is ensured by inserting casts as shown in their previous work for functional languages [3]. Ten years later, many systems support the gradual addition of types to untyped object-oriented programs.[1]

Despite its age and popularity, faithful realizations of Siek and Taha's elegant idea have proved to be surprisingly elusive, one possible reason being that the original paper did not consider state. The combination of mutable state, aliasing and subtyping complicates enforcement strategies as one must consider situations where an object is being accessed at different types. While several solutions have been proposed to address this, the performance implications of the implementation strategies of these solutions seem 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 some of the forces that have influenced existing systems and discuss the implication of key design decisions. While there are significant challenges that, in the end, may prevent adoption of some of the more ambitious type systems, there are are also opportunities for improving on existing techniques. This paper also aims to lay the agenda for future investigations.

[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] *Sound* type systems reports order of magnitude 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. Most systems lack rigorous evaluations.

[3] Dart, Typescript and Hack use unchecked modes for production, all type errors will not be reported.

To capture the essence of gradual typing for objects and to highlight the challenges implementers face, we present translations of representative subsets of gradually typed languages into a common target language. Targeting the same language lets us reason about the type systems in the same framework and allows for comparison. But which language should we target? The language should avoid linguistic clutter while expressing key object-oriented features directly rather than by encoding. We propose to target a typed object-oriented language inspired by modern language runtime such as the Java Virtual Machine and the CLR in .Net. They have a static type system with classes and subtyping but they also allow for dynamic resolution of method dispatch.[4] As well as reasonably efficient implementations, and in both environments, primitive types can be unboxed.

A contribution of this work is the design of KafKa, a statically typed object calculus. KafKa is class based (with an explicit class table K), with mutable state (a heap address a refers to an object, each field f is reified into a pair of getter and setter methods), and allows the dynamic generation of wrapper classes (by update of the class table K and allocation of new objects a). Methods can be statically resolved, denoted by a simple call a.m(x), or can be dynamically resolved, denoted by a dynamic call a@m(x). The KafKa type system has two types, a structural type C, allow for recursive typing and stored in a class table K, and the dynamic type $\star$. Types in general are referred to as t.

The heart of any gradual type system implementation is the explicit casts that are inserted at type boundaries. Two different *structural* casts are built-in to KafKa; they only inspect the structure of objects. The subtype cast $<$t$>$ a is a structural cast that checks if the object references by a is a subtype of type t. The shallow cast $\prec$ t $\succ$ a merely checks that the object has all the methods of type t without looking at their signatures. To support some of the more complex type systems, KafKa is extended by *generative* casts which create new wrapper classes. The behavioral cast ◀ t ▶ a generates a wrapper object that monitors that methods invoked on a abide by the interface of type t. The monotone cast ◁t▷a returns a wrapper monitoring all invocations to the object referenced by a, furthermore this wrapper can only be refined by removing occurrences of $\star$ in its type.

Another contribution is the translation of five type systems representative of the main strains of gradual typing for objects. Each gradually typed expression $\underline{\mathsf{e}} : \underline{\mathsf{T}}$ in the source is translated to a well-typed KafKa term, e : T, each type $\underline{\mathsf{T}}$ has a corresponding KafKa type T, and, similarly, each expression $\underline{\mathsf{e}}$ has an equivalent e. While soundness in KafKa is straightforward (except for generative casts), soundness of the source gradual type systems is more interesting. In KafKa, a well-typed program can only get stuck at a cast or a dynamically resolved call. With gradual types, an expression $\underline{\mathsf{x}}.\mathsf{m}(\underline{\mathsf{x}'})$ where $\underline{\mathsf{x}}$ is declared to be of class $\underline{\mathsf{C}}$ can have significantly different behavior depending on choices made while designing the gradual type system. TypeScript has *optional types*; a well-typed program can get stuck at any method call as $\underline{\mathsf{C}}$ translates to $\star$. Thorn has *concrete types*; a well-typed program will not get stuck at statically resolved method calls, and $\underline{\mathsf{C}}$ maps to itself C. Typed Racket has *promised types*; a well-typed program will not get stuck at a call to m, because $\underline{\mathsf{x}}$ refers to an object, or wrapper, that implements m. Wrappers may fail if their target does not behave like a $\underline{\mathsf{C}}$. Finally we describe the monotonic and transient variants of Reticulated Python which have somewhat surprising semantic properties.

[4] Both runtime support reflective invocation based on method names. Dynamic resolution was added to C# in version 4.0 [14]. In Java, invokedynamic allows for custom method dispatch.

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 [17] which led to Bracha's notion of pluggable types [18]. For him, types are solely to catch errors at compile-time, they should never affect 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 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 [19] design that later became Typed Racket is strongly influenced by 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 constraint, another constraint was to prevent surprises – a variable annotated as a C should behave as a C. Any change in behavior 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 $\mathsf{t} \sim \mathsf{t}'$ which states that types that agree on non-$\star$ position are compatible. In [1] they extended their result to a stateless object calculus and combined consistency with structural subtyping. Extending the approach to mutable objects proved challenging. Reticulated Python [9] attempts to find a compromise between soundness and efficiency. The language has three modes: *guarded* mode behaves as Racket with contracts applied to values, *transient* mode performs first order checks before each call, and *monotonic* mode is similar to the guarded mode except in there are no wrapper chains and casts only remove occurrences of $\star$ from an object's type.

Other noteworthy systems include Gradualtalk [8], C# 4.0 [4], Thorn [12], 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 with type $\star$. This means C# has a dynamic sublanguage that allows developer to write unchecked code, but it also has a strongly type 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.[5] 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.

	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

Figure 1 Overview of implemented 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 the 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. 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

[5] 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.

values. A fitting analogy are the stack traces printed by Java when a program terminates abruptly. Developers working in, e.g, C++ must run their program in a debugger to obtain the same information. Stack traces have little run-time cost because they piggyback on another feature, namely precise exceptions, which does come at a price as it inhibits some compiler optimizations. It is likely that recording blame is costly, but there is no data on how much implementations pay for it.

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 Gradual Typing in Practice

"Oh, let me see your beauty when the witnesses are gone"

While gradual type systems for objects come in many shapes and sizes, with different constraints and complexity, as well as their own strength and weaknesses. This section gives a brief introduction to the five representative systems that we are studying in this paper. We conclude with a discussion of the trace preservation property and how the systems differ in terms of catching errors. Rather than speaking directly to the source systems, we present their KafKa modelization. The details of this modelization are the topic of the rest of the paper, so this section can be read as a preview. We leave blame for future work, and discuss semantics in terms of their translation to a typed class-based calculus with structural subtyping.

3.1 TypeScript

Typescript [14] is a backwards compatible extension of JavaScript with classes and type annotations. Type equivalence is structural and subtyping of recursive types is supported (types arise from class declarations). Missing annotations are treated as $\star$. The role of types is to catch simple errors, such as misspelt method names, as well as assisting IDEs. A typical example is shown here. The first expression is ill-typed because method `o` does not exist, the second is erroneous as it provides an instance of `A` where `C` is expected. The third expression is statically correct as the instance of `A` is cast to a `C` by the `n` method.

```
class A {
   m(x: C): A { this }
   n(x: *): C { x } }

new A().o(new C()) // ERR wrong method
new A().m(new A()) // ERR wrong type
new A().m(new A().n(new A())) // OK
```

Well-typed code is translated to plain JavaScript with all types erased and methods resolved dynamically. In KafKa, one would translate the class definition and the last expression:

```
class A {
   m(x: *): * { this }
   n(x: *): * { x } }
new A()@m(new A()@n(new A()))
```

Dynamically resolved calls (written `@`) are calls that may fail because the receiver need not have the requested method. Method signatures see their types set to ⋆. The designers of TypeScript saw unsoundness as a way to ensure, (*a*) 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 (*b*) an insurance for backwards compatibility, as ignoring types means all browsers can run TypeScript code – with no additional overhead.

3.2 Thorn

Thorn [12] is an object-oriented language with multiple inheritance, nominal type equivalence, and a combination of optional and concrete types. Optional types (written `?T`) are translated to the ⋆ type with dynamically resolved method invocation, as in TypeScript. Concrete types (written `T`) behave as one would expect: a variable `x:C` is guaranteed to refer to an instance of `C` or a subtype thereof. Consider the following program with calls to concretely typed method `n` and optionally typed `m`, and assume that `D` is not a subtype of `C`.

```
class A {
   m(x: ?C): ?C { x }
   n(x:  C):  C { x } }

new A().m( new D() ) // OK -- warning
new A().n( new D() ) // ERR -- D !<: C
new A().m( new A.n(new C()) // OK
new A().n( new A.m(new C()) // Cast
```

The first call to `m` is allowed because the argument of `m` is treated as type ⋆, and a warning is emitted. The second call is ruled incorrect as a concrete type is expected for `n`. The third call is allowed as a `C` is a subtype of `?C`. The last call results in a cast being inserted as `?C` cannot be guaranteed to hold an instance of `C` or a subtype. The translation of Thorn to KafKa turns optional types to ⋆ and inserts structural casts to `C` when an expression of type ⋆, or of type `?C` is assigned to a `C`. So class `A` would translate to:

```
class A {
   m(x:  *):  * { x }
   n(x:  C):  C { x } }
```

3.3 Transient Python

The Transient variant of Reticulated Python [9] aims for soundness with a predictable cost model. The declared types of arguments are defensively checked in every method, such as in method `m`. Method `m` expects an instance of class `C`, but it can be called with a value of type ⋆ at any point, forcing `m` to check its arguments at every invocation.

```
class A {
   m(x: C): D { x.n(new C()) }
   n(x: *): * { x } }

new A().m( new A().n( new C() ) ) // OK
new A().m( new A().n( new D() ) ) // Runtime ERR
```

Transient Python deems both expressions well-typed. However, the second is going to end up with a type error as a `D` is passed to a method expecting a `C`. When translating to KafKa, all types are erased and casts are inserted on method entry and prior to returning.

```
class A {
  m(x: *): * { ≺ D ≻ (≺ C ≻ x)@n(new C()) }
  n(x: *): * { x } }
```

The translated class `A` has a check to validate that the argument to `m` has all the methods defined by `C`. For this we use KafKa's *shallow structural cast*, written ≺ `C` ≻ `x`, which only checks for the presence of method names, not for the conformance of type signatures.

3.4 Typed Racket

Typed Racket has an expressive object system with first-class classes [13], which we restrict to the core of the system, a class-based object system with structural subtyping. Classes are either typed, in which case all variables have types different from ⋆, or untyped, where all types are ⋆. In the following example, `A` is untyped, it constructs an instance of the typed class `B` and invokes method `n` with `x`, a variable of type ⋆.

```
class A {
  m(x: *): * { new B().n(x) } }
class B {
  n(x: B): C { x.n(x) } }

new A().m( new A())
```

Class `B` expects `x` to be of type `B`, yet it will be given an `A`. Typed Racket mediates at boundaries between typed and untyped code by inserting wrappers around exchanged values. These wrappers ensure that the values behave according to their *promised* types. The translation to KafKa is such that typed code remains untouched, untyped code is extended with *behavioral generative casts* at creation of instances of typed classes. The body of `m` becomes `(◀ ⋆ ▶new B())@n(x)`. The instance of `B` is cast to ⋆, and the call is dynamically resolved. The cast creates a new wrapper class that has all of the methods of `B` accepting untyped arguments as well as typed arguments. The wrapper generated by the cast is (roughly) as follows:

```
 class BW {
  that : B
  n(x: *): * { ◀ ⋆ ▶ that.n( ◀ B ▶ ≺ B ≻ x ) }
  n(x: B): C { that.n(x) } }
```

Class `BW`, a fresh name, is a subtype of `B` (in KafKa, subtyping is defined only over typed methods). The wrapper has a field that which refers to the instance of `B`. When it is called from a typed context, all it does is forward calls to the target. In an untyped context, the wrapper will cast the argument to the promised type and cast results back to ⋆. Each of these casts introduces new wrappers. A shallow structural cast is used to catch obvious mismatches early. The key properties of this translation are that typed code can rely on the presence of methods of the right type (and thus use static method resolution) but calls can still fail at casts within the wrapper. An easily overlooked, but significant, feature of Racket is that the this variable is wrapped by any wrapper(s) applied at the call site.

3.5 Monotonic Python

Monotonic Python [21] aims to be sound while avoiding wrapper chains and supporting object mutation. This is achieved via associating an effective type to every object and updating the effective type on casts. Thus, an instance of class `A` can be cast to `B` in Monotonic.

```
class A {
   f : *
   m(x: *): * { this.f } }
class B {
   f : B
   m(x: B): B { this.f } }
```

If A is used as a linked data structure, casting the head object to B will update its effective type to match B, *and will recursively follow field* f *and update all of the objects that are reachable through it.* The name of the semantics comes from the fact that only a finite number of casts can change the effective type of an object, as Monotonic only allows casting to types that have fewer occurrences of $\star$. Subtyping is defined using a combination of structural equivalence and consistency. This ensures that in case of aliasing, references that are of type A can safely point to the updated type B. The main design issue is that this causes untyped code to be "brittle," since any cast that changes the effective type of an object will forever brand the object with that type. As a result, newly added yet distant types, or slightly changed library code, can cause substantial knock-on effects under the monotonic semantics, breaking previously working untyped code. The translation to KafKa involves creating a single wrapper that encodes the effective type of the object. Monotonic generative casts are inserted, where appropriate, to create wrapper classes that further specialize and enforce the effective type, replacing the original, more dynamic, wrapper.

3.6 Trace preservation

One outcome of our work is showing how *all* of the gradual type systems are observationally distinct. Fig. 2 presents four litmus tests that are sufficient to distinguish the five type systems being studied in this paper. Each litmus test is a program composed of a class table and a main expression. The programs are written in KafKa syntax but equivalent programs can be expressed in each of the target languages. All programs are well-typed in each of the respective type systems and give raise to different runtime errors. Under TypeScript semantics, all programs run to completion without getting stuck. With Thorn, all litmus programs fail at a structural cast from A to I. For the other languages the situation is as follows:

- **L1:** Fails because A is not a subtype of I, the failure is at a shallow cast because A does not have all the methods of I.
- **L2:** Succeeds in Monotonic/Transient/Racket because shallow casts of A to I go through.
- **L3:** Fails in Monotonic because the same object is being cast to two different types, I and J.
- **L4:** Fails in Monotonic because the object refereed to by x in method s is cast to I. This updates the effective type of field f to D. However, the assignment in method m tries to give it an A. This fails in Racket because the this field is wrapped with an I type, and the assignment to f does not respect that type.

```
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)}}
```

(Litmus test 1)

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

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

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

(Litmus test 2)

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

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

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

class E {f:I g:J}

class T {
 t(x:*):* {
   new E(x,x)}}
```

(Litmus test 3)

```
class A {
 f:*
 m(x:A):A {
  this.f(new A(new C())
      )}}


class I {
 f:D
 m(x:I):I {this}}


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

(Litmus test 4)

```
class C {
  n(x:C):C {this}}

class D {
  o(x:D):D {this}}
```

(Auxiliary classes)

```
new T()@t(new A())
```

(Program 1-3)

```
new T()@t(
    new A(new D()))
```

(Program 4)

	L1	L2	L3	L4
Thorn				
Typed Racket		✓	✓	
Monotonic RetPy		✓		
Transient RetPy		✓	✓	✓
TypeScript	✓	✓	✓	✓

Figure 2 Semantic litmus tests.

4 KafKa: A Core Calculus

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

The basis of our formal approach is the KafKa calculus. At its heart, KafKa is a class-based, object-oriented, structurally typed language with dynamic dispatch, modeled off of common compilation targets such as the JVM's Java Bytecode or the .NET CLR's Common Intermediate Language, and taking inspiration from C#'s approach to gradual typing [4]. The syntax of KafKa is given in Fig. 3. Meta-variables C, D and E range over class names, x ranges over variable names, m and f range over methods and fields respectively. Finally, this is a distinguished identifier representing a method receiver, while that is a distinguished field name used in wrapper classes. Expressions include object creation, $\mathbf{new}\ \mathsf{C}(\overline{\mathsf{e}})$, field reads, $\mathsf{e.f()}$, field updates, $\mathsf{e.f(e)}$, statically resolved method invocation, $\mathsf{e.m(e)}$, dynamically resolved method invocation, $\mathsf{e@m(e)}$, subtype casts, $<\mathsf{t}>\ \mathsf{e}$, and shallow casts, $\prec \mathsf{t} \succ \mathsf{e}$. Types consist of class names and the dynamic type, written $\star$. Class definitions have a class name and (possibly empty) sequences of field definitions and method definitions, $\mathbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\}$. Field definitions consist of a field and its type, $\mathsf{f : t}$. Method definitions have (for simplicity) a single argument and an expression $\mathsf{m(x\colon t)\colon t}\ \{\mathsf{e}\}$. Fields are read by calling getter methods and updated by setter methods, these can be defined as, respectively, $\mathsf{f()\colon t}\ \{\mathsf{e}\}$ and $\mathsf{f(x\colon t)\colon t}\ \{\mathsf{e}\}$.

$$
\begin{array}{llll}
\mathsf{e} ::= \mathsf{x} & \mid\ \mathsf{this} & \mid\ \mathsf{that} \\
\quad\mid\ \mathbf{new}\ \mathsf{C}(\overline{\mathsf{e}}) & \mid\ \mathsf{e.f()} & \mid\ \mathsf{e.f(e)} \\
\quad\mid\ \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{e}) & \mid\ \mathsf{e@m(e)} & \mid\ <\mathsf{t}>\mathsf{e} \\
\quad\mid\ \prec \mathsf{t} \succ \mathsf{e} & \mid\ \mathsf{a}
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{k} & ::= & \mathbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\} \\
\mathsf{md} & ::= & \mathsf{m(x\colon t)\colon t}\ \{\mathsf{e}\} \mid \mathsf{f(x\colon t)\colon t}\ \{\mathsf{e}\} \mid \mathsf{f()\colon t}\ \{\mathsf{e}\} \\
\mathsf{t} & ::= & \star \mid \mathsf{C} \\
\mathsf{fd} & ::= & \mathsf{f : t}
\end{array}
$$

Figure 3 KafKa Syntax.

4.1 Static Semantics

A well-formed program, denoted $\mathsf{e}\ \mathsf{K}\ \checkmark$, consists of an expression e and a class table K where each class k in 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. A class that defines a field f is not allowed to have getter or setter methods of that name. A class can have at most two definitions for any method m, one typed $\mathsf{m(x\colon C)\colon D}\ \{\mathsf{e}\}$ and one untyped $\mathsf{m(x\colon \star)\colon \star}\ \{\mathsf{e}\}$. The type judgments, $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e : t}$, are mostly unsurprising, with the following rules for method calls and casts.

$$
\textsc{w6}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e : C} \qquad \mathsf{m(t')\colon t''} \in \mathsf{mtypes(C,K)} \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e' : t'}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.m(e') : t''}}
\qquad
\textsc{w7}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e'} : \star}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e@m(e')} : \star}
$$

$$
\textsc{w9}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e : t'}}{\Gamma\,\sigma\,\mathsf{K} \vdash <\mathsf{t}>\mathsf{e : t}}
\qquad
\textsc{w10}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e : t'}}{\Gamma\,\sigma\,\mathsf{K} \vdash \prec \mathsf{t} \succ \mathsf{e} : \star}
$$

Method calls use *syntactic* disambiguation to select between typed and untyped methods. The dynamically resolved call places no requirements on the receiver or argument, and returns type $\star$. The subtype cast rule W9 states that result is of the cast type, but it does not preclude silly casts. Rule W10 describes the semantics of the shallow casts, which merely asserts the presence of method names, without making any statements about the type of the object being cast. The subtyping relation, $\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$, allows for recursive structural subtyping. The distinctive feature of the subtype relation is that fields appears in the type signature as getter/setter pairs with the exception of the that field that is hidden from the class type signature. The complete set of rules appear in Appendix section A.

4.2 Dynamic Semantics

The small operational step operational semantics for KafKa appears in Fig. 4. A configuration is a triple $\mathsf{K}\ \mathsf{e}\ \sigma$, where K is a class table, e is an expression and σ is a heap mapping addresses ranged over by a to objects denoted $\mathsf{C}\{\bar{\mathsf{a}}\}$. A configuration evaluates in one step to a new configuration, $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$, the new configuration may include a new class table built by extending the previous table with new classes. Field access either reads a field from the object (if the class has a field definition) or invokes a getter function (if the class has a getter method of the field name), and setters are treated similarly. Calling forms specify the typing, typed methods are invoked with static calls and untyped methods are invoked with dynamically resolved calls. Casts to $\star$ are trivially satisfied while subtype casts to C check that the runtime object is an instance of a subtype of C, while shallow casts check inclusion of method names. Evaluation contexts are deterministic and enforce a strict evaluation order.

5 Translating Gradual Types

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

This section gives semantics to the five gradual type systems of interest. In each case, the semantics of the source languages are defined by translation to KafKa programs. Source and target share most of their syntax, and when we need to distinguish them, we will use

$$\boxed{\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'} \quad \mathsf{e}\ \sigma \text{ evaluates to } \mathsf{e}' \text{ in a step}$$

$$\begin{array}{lll}
\mathsf{K}\ \textbf{new}\ \mathsf{C}(\bar{\mathsf{a}})\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\bar{\mathsf{a}}\}] & \mathsf{a}' \text{ fresh} \\
\mathsf{K}\ \mathsf{a.f}()\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a/this}]\mathsf{e}\ \sigma & \mathsf{f}() : \mathsf{t}\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a.f}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \mathsf{f}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a.f}()\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma & \mathsf{read}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{K}) = \mathsf{a}' \\
\mathsf{K}\ \mathsf{a.f}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}'\ \sigma' & \mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{a}', \mathsf{K}) = \sigma' \\
\mathsf{K}\ \mathsf{a.m}_{\mathsf{t}\rightarrow\mathsf{t}'}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}'\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a@m}(\mathsf{a}')\ \sigma & \rightarrow \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \sigma & \mathsf{m}(\mathsf{x} : \star) : \star\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C}) \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ {<}\star{>}\ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \\
\mathsf{K}\ {<}\mathsf{D}{>}\ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \mathsf{K} \vdash \mathsf{C} <: \mathsf{D} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ \prec\star\succ\ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \\
\mathsf{K}\ \prec\mathsf{D}\succ\ \mathsf{a}\ \sigma & \rightarrow \mathsf{K}\ \mathsf{a}\ \sigma & \mathsf{names}(\mathsf{K}(\mathsf{C})) \supseteq \mathsf{names}(\mathsf{K}(\mathsf{D})) \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\bar{\mathsf{a}}\} \\
\mathsf{K}\ \mathcal{E}[\mathsf{e}]\ \sigma & \rightarrow \mathsf{K}'\ \mathcal{E}[\mathsf{e}']\ \sigma' & \mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'
\end{array}$$

$$\begin{array}{lllllllll}
\mathcal{E} ::= \mathcal{E}.\mathsf{f}() & \mid \mathcal{E}.\mathsf{f}(\mathsf{e}) & \mid \mathsf{a.f}(\mathcal{E}) & \mid \mathcal{E}.\mathsf{m}(\mathsf{e}) & \mid \mathsf{a.m}(\mathcal{E}) & \mid \mathcal{E}@\mathsf{m}(\mathsf{e}) & \mid \mathsf{a@m}(\mathcal{E}) \\
\quad \mid\ {<}\mathsf{t}{>}\ \mathcal{E} & \mid \prec\mathsf{t}\succ\ \mathcal{E} & \mid \textbf{new}\ \mathsf{C}(\bar{\mathsf{a}}\ \mathcal{E}\ \bar{\mathsf{e}}) & \mid \square
\end{array}$$

Figure 4 KafKa Semantics.

underline to denote source terms. Casts are not used in the source languages, but are inserted by the translations. The general approach in each translation is the following, starting with a program $\underline{\mathsf{K}\ \mathsf{e}}$, we define a translation to a KafKa configuration $\mathsf{K}\ \mathsf{e}$.

$$\boxed{\begin{array}{lcrcr} \underline{\mathsf{K}\ \mathsf{e}} & \hookrightarrow_p & \mathsf{K}\ \mathsf{e} & \longrightarrow & \mathsf{K}'\,\mathsf{e}'\,\sigma \\ & & \mathsf{K}\ \mathsf{e}\ \checkmark & & \cdot\,\sigma\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t} \\ & & \cdot\ \cdot\ \mathsf{K} \vdash \mathsf{e} : \mathsf{t} & & \end{array}}$$

Each translation guarantees a few key properties. A translated program is well-formed, a well-formed program will reduce or get stuck at a cast or a dynamically resolved call.

5.1 TypeScript

To model TypeScript, we use a very simple translation mechanism, where every type becomes $\star$ and every call becomes a dynamic call. We proceed through translation top-down, translating classes and the main expression. Beyond the top level translation, we have two judgments that perform cast insertion on expressions, following Pierce and Turner [22], the synthetic cast insertion judgment $\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}}$, which states that against context Γ and class table $\underline{\mathsf{K}}$, expression $\underline{\mathsf{e}}$ translates to e' producing source type $\underline{\mathsf{t}}$, and the analytic judgment $\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'$, which ensures that the translated expression e' will model source type $\underline{\mathsf{t}}$, coming from source expression $\underline{\mathsf{e}}$. Inside of this bidirectional framework, we then have a fairly typical set of translation judgments.

The synthetic case of translation is uninteresting, with the exception of TSS4, which allows access to fields via the self-reference, whose type is always known. Of more interest are the analytic rules: TSA1 requires the interaction between typed parts of the program to be well-typed, but TSA2 and TSA3 allow untyped values to be passed into typed references without any extra checks, and are what causes TypeScript to be unsound. The translation produces well-typed terms as it erases all types in the source during translation. The translation is weak due to the type erasure, as it befits TypeScript's optional and unsound nature.

TSP

$$\frac{\underline{\mathsf{K}} \vdash \underline{\mathsf{K}} \;\hookrightarrow_c\; \mathsf{K}' \qquad \underline{\mathsf{K}\ \cdot} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}}}{\underline{\mathsf{e}\,\mathsf{K}} \;\hookrightarrow_p\; \mathsf{e}'\,\mathsf{K}'}$$

TSC1

$$\frac{\begin{array}{c}\underline{\mathsf{md}} \in \underline{\overline{\mathsf{md}}} \implies \underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{md}} \;\hookrightarrow_m\; \mathsf{md}' \wedge \mathsf{md}' \in \overline{\mathsf{md}'} \\ \underline{\mathsf{f}: \mathsf{t}} \in \underline{\overline{\mathsf{fd}}} \implies \mathsf{f}: \star \in \overline{\mathsf{md}'} \\ \underline{\mathsf{K}} \vdash \underline{\mathsf{K}'} \;\hookrightarrow_c\; \mathsf{K}''\end{array}}{\underline{\mathsf{K}} \vdash \underline{\textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\}\ \mathsf{K}'} \;\hookrightarrow_c\; \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}'}\ \overline{\mathsf{md}'}\,\}\ \mathsf{K}''}$$

TSC2

$$\frac{}{\underline{\mathsf{K}} \vdash \underline{\cdot} \;\hookrightarrow_c\; \cdot}$$

TSM

$$\frac{\underline{\mathsf{K}\ \mathsf{this}: \mathsf{C}\ \mathsf{x}: \mathsf{t}} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}'} \hookrightarrow \mathsf{e}'}{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{m}(\mathsf{x}: \mathsf{t}): \mathsf{t}'\ \{\mathsf{e}\}} \;\hookrightarrow_m\; \mathsf{m}(\mathsf{x}: \star): \star\ \{\mathsf{e}'\}}$$

TSS1

$$\frac{\underline{\Gamma}(\underline{\mathsf{x}}) = \underline{\mathsf{t}}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{x}} \hookrightarrow \mathsf{x} \Uparrow \underline{\mathsf{t}}}$$

TSS2

$$\frac{\begin{array}{c}\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_2 \Uparrow \underline{\mathsf{C}} \\ \underline{\mathsf{m}}(\underline{\mathsf{t}}): \underline{\mathsf{t}'} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}}) \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}'_1} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'_2\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{m}(\mathsf{e}'_1)} \hookrightarrow \mathsf{e}_2@\mathsf{m}(\mathsf{e}'_2) \Uparrow \underline{\mathsf{t}'}}$$

TSS3

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}_2 \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}'_1} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}'_2}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1@\mathsf{m}(\mathsf{e}'_1)} \hookrightarrow \mathsf{e}_2@\mathsf{m}(\mathsf{e}'_2) \Uparrow \underline{\star}}$$

TSS4

$$\frac{\begin{array}{c}\underline{\Gamma}(\underline{\mathsf{this}}) = \underline{\mathsf{C}} \\ \underline{\mathsf{f}}(\underline{\mathsf{t}}): \underline{\mathsf{t}} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}}) \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{this.f}(\mathsf{e})} \hookrightarrow \mathsf{this.f}(\mathsf{e}') \Uparrow \underline{\star}}$$

TSS5

$$\frac{\underline{\Gamma}(\underline{\mathsf{this}}) = \underline{\mathsf{C}} \qquad \underline{\mathsf{f}}(): \underline{\mathsf{t}} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}})}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{this.f}()} \hookrightarrow \mathsf{this.f}() \Uparrow \underline{\star}}$$

TSS6

$$\frac{\overline{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\textbf{new}\ \mathsf{C}(\overline{\mathsf{e}})} \hookrightarrow\!<\!\star\!>\ \textbf{new}\ \mathsf{C}(\overline{\mathsf{e}'}) \Uparrow \underline{\mathsf{C}}}$$

TSA1

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}'} \qquad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'}$$

TSA2

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\star}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'}$$

TSA3

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{C}}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}'}$$

Figure 5 TypeScript translation.

5.2 Thorn

To model Thorn we extend the syntax of source terms with an "unsound" type written ?C, which has its own subtyping relation, and compiles down to $\star$. This unsound type still enforces the same static requirements as its sound counterpart, but generates no runtime casts. Over this new type, we define a subtyping relation, $\mathsf{t}\ \underline{<:}\ \mathsf{t}$. The relation is identical to KafKa subtyping except in that ?C $\underline{<:}$?D and C $\underline{<:}$?D both hold if C $\underline{<:}$ D. The synthetic cases describe the traditional "bottom-up" style of typing judgment. Each case produce an equivalent KafKa term for the given Thorn expression under the KafKa type that matches the Thorn type. Rules THS1, THS2, THS5, and THS6 are all effectively direct translations of the traditional typing rules for variables, calls, dynamic invocation, and new, respectively. However, two rules, THS3 and THS4, break this pattern. These rules are applied when the receiver type is an unsound type and case analyze is performed on the declared return type. In THS3, the return type is a concrete type, or a sound type, from an unsound type, and therefore requires an insertion of a runtime cast, and the alternative case is shown in THS4. It is important to note that both THS3 and THS4 produce dynamic calls on the receiver, despite the receiver having a source language type. This is because Thorn's unsound types are translated to $\star$, which causes all calls to be dynamic.

THS1

$$\frac{\underline{\Gamma}(\underline{\mathsf{x}}) = \underline{\mathsf{t}}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{x}} \hookrightarrow \mathsf{x} \Uparrow \underline{\mathsf{t}}}$$

THS2

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{C}} \qquad \underline{\mathsf{f}}() : \underline{\mathsf{t}} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}})}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e.f}()} \hookrightarrow \mathsf{e}'.\mathsf{f}() \Uparrow \underline{\mathsf{t}}}$$

THS2'

$$\frac{\begin{array}{c}\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_2 \Uparrow \underline{\mathsf{C}} \\ \underline{\mathsf{f}}(\underline{\mathsf{t}}) : \underline{\mathsf{t}} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}}) \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}'_1} \Downarrow \underline{\underline{\mathsf{t}}} \hookrightarrow \mathsf{e}'_2\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{f}(\mathsf{e}'_1)} \hookrightarrow \mathsf{e}_2.\mathsf{f}(\mathsf{e}'_2) \Uparrow \underline{\mathsf{t}}}$$

THS2"

$$\frac{\begin{array}{c}\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_2 \Uparrow \underline{\mathsf{C}} \\ \underline{\mathsf{m}}(\underline{\mathsf{t}}) : \underline{\mathsf{t}'} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}}) \qquad \overline{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\mathsf{t}_1} \hookrightarrow \mathsf{e}_4}\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{m}(\mathsf{e}'_1)} \hookrightarrow \mathsf{e}_2.\mathsf{m}(\mathsf{e}'_2) \Uparrow \underline{\mathsf{t}'}}$$

THS3

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_3 \Uparrow \underline{?\mathsf{C}} \qquad \underline{\mathsf{m}(\mathsf{t}_1) : \mathsf{D} \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})} \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\mathsf{t}_1} \hookrightarrow \mathsf{e}_4}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)} \hookrightarrow <\mathsf{D}> \mathsf{e}_3@\mathsf{m}(\mathsf{e}_4) \Uparrow \underline{\mathsf{D}}}$$

THS4

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_2 \Uparrow \underline{?\mathsf{C}} \qquad \underline{\mathsf{m}}(\underline{\mathsf{t}}) : \underline{?\mathsf{D}} \in \mathsf{mtypes}(\underline{\mathsf{C}}, \underline{\mathsf{K}}) \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}'_1} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'_2}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{m}(\mathsf{e}'_1)} \hookrightarrow \mathsf{e}_2@\mathsf{m}(\mathsf{e}'_2) \Uparrow \underline{?\mathsf{D}}}$$

THS5

$$\frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}_3 \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}_4}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1@\mathsf{m}(\mathsf{e}_2)} \hookrightarrow \mathsf{e}_3@\mathsf{m}(\mathsf{e}_4) \Uparrow \underline{\star}}$$

THS6

$$\frac{\overline{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\textbf{new}\ \mathsf{C}(\overline{\mathsf{e}})} \hookrightarrow \textbf{new}\ \mathsf{C}(\overline{\mathsf{e}'}) \Uparrow \underline{\mathsf{C}}}$$

Figure 6 Thorn translation.

5.3 Transient Python

The Transient semantics of Fig. 7 illustrates the use of the structural shallow cast, $\prec \mathsf{t} \succ \mathsf{e}$. The Transient semantics makes guarantee about what methods are avaible in the argument and return value of each typed method. The translation require several additional rules in order to generate well-typed terms. Rule TPM1 uses an auxiliary class A which is a tuple for sequencing. It first performans a shallow cast of the argument x to C and then evaluates the body of the method. (The call to f return the second field in A, i.e., body's value). All type annotations are erased and replaced by $\star$. Every call, typed or untyped, in Transient becomes untyped. Every typed method call must be guarded with a cast to ensure that the return type of the method is correct, as shown in TPG2. The lack of statically typable calls in the Transient semantics means that a pure implementation of transient would entail no fields ever being accessed using the setter or getter methods, as KafKa forbids accessing fields through dynamic call sites. To overcome this, rule TPG4 is added, which allows the definition of self-referential field access. In a transient system, the only type that is known is the type of this. As a result, accesses to fields have to go through user-defined field accessor methods. Rule TPR2 provides consistency for Transient. In gradual type systems with consistency, consistency is used to conclude an analytic judgment, as consistency allows the system to break type guarantees, adding and removing types from sub-parts of a program. Consistency within KafKa is defined in a later section, detailing the monotonic cast semantic and the translation for the monotonic system, but is used in the transient system to allow typed and untyped code to interact. When consistency is used, a check is inserted to ensure that all the methods that are required are declared on the provided type, but the types of the methods are ignored, as they are the responsibility of the callee to check.

TPM1

$$\frac{\underline{K\ \mathsf{this}\colon C\ x\colon \star} \vdash \underline{e} \Downarrow \underline{\star} \hookrightarrow e'}{\underline{K\ C} \vdash \underline{m(x\colon \star)\colon \star\,\{e\}} \ \hookrightarrow_m\ m(x\colon \star)\colon \star\,\{e'\}}$$

TPM2

$$\frac{t \neq \star \qquad \underline{K\ \mathsf{this}\colon C\ x\colon t} \vdash \underline{e} \Downarrow \underline{t'} \hookrightarrow e'}{\underline{K\ C} \vdash \underline{m(x\colon C)\colon C'\ \{e\}} \ \hookrightarrow_m\ m(x\colon \star)\colon \star\,\{<\star>\ \mathbf{new}\ A(\prec C \succ x, e).f()\}}$$

TPG1

$$\frac{E(x) = t}{\underline{K\ E} \vdash \underline{x} \hookrightarrow \prec t \succ x \Uparrow \underline{t}}$$

TPG2

$$\frac{\begin{array}{cc} & \underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \\ m(t_1) : t_2 \in \mathsf{mtypes}(C, K) & \overline{\underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4} \end{array}}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow \prec t_2 \succ (<\star>\ e_3)@m(<\star>\ e_4) \Uparrow \underline{t_2}}$$

TPG3

$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{\star} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{\star} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m(e_4) \Uparrow \underline{\star}}$$

TPG4

$$\frac{\begin{array}{c} E(\mathsf{this}) = C \qquad f(\overline{t_1}) : t_2 \in \mathsf{mtypes}(C, K) \\ \underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t_1} \hookrightarrow e' \end{array}}{\underline{K\ \Gamma} \vdash \underline{\mathsf{this}.f(\bar{e})} \hookrightarrow \mathsf{this}.f(e') \Uparrow \underline{\star}}$$

TPG5

$$\frac{\overline{\underline{K\ E} \vdash \underline{e_1} \Downarrow \underline{t} \hookrightarrow e_2} \qquad \mathbf{class}\ C\,\{\,\overline{f\colon t}\ \overline{md}\,\}}{\underline{K\ \Gamma} \vdash \underline{\mathbf{new}\ C(\overline{e_1})} \hookrightarrow \mathbf{new}\ C(\overline{e_2}) \Uparrow \underline{C}}$$

TPR1

$$\frac{\underline{K\ E} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash t' <: t}{\underline{K\ E} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$$

TPR2

$$\frac{\underline{K\ E} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash t' \sim t}{\underline{K\ E} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow \prec t \succ e'}$$

Figure 7 Transient translation.

5.4 Typed Racket

Our model of Typed Racket does not preserve the "macro" model of gradual typing of the source language, as it does not effect the fundamental semantics of the typing system. Typed Racket requires a new cast, a *generative cast*, which is sufficiently different from what we have seen so far that we will extend the syntax and semantics of KafKa to support it. For an expression e and type t, the cast $\blacktriangleleft t \blacktriangleright e$ depicts the behavioural cast, and the typing and semantic rules are next.

Static Typing

WB1

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

Dynamics

$$K\ \ \blacktriangleleft t \blacktriangleright a\ \sigma \!\dashrightarrow K'\ a'\ \sigma' \qquad \mathsf{behcast}(a, t, \sigma, K) = K'\ a'\ \sigma'$$

$$\mathcal{E} ::= \ldots \ \mid\ \blacktriangleleft t \blacktriangleright \mathcal{E}$$

The definition of $\mathsf{behcast}(a, t, \sigma, K)$ are presented in the next subsection. Observe that, despite not checking the value, we assert that the type is correct, and, notice that the cast returns a new object reference a′ and an updated class table K′. This is a wrapper object that is, being construction, of the correct type. The wrapper's class, added in K′, has code to ensure that the value behaves appropriately. The translation is fairly simple as a result of the strong type invariants that are exposed to the static code. TRA2 and TRA3 are the most important cases of cast insertion; they capture values crossing type boundaries. In the case of an untyped object gaining a type assertion (TRA2), a shallow cast ensures that the value has all of the required method names before applying the behavioural cast.

$$\text{TRA1}\quad \frac{\underline{K}\,\underline{\Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash \underline{t'} <: \underline{t}}{\underline{K}\,\underline{\Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'} \qquad\qquad \text{TRA2}\quad \frac{\underline{K}\,\underline{\Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad \underline{t} \neq \underline{t'}}{\underline{K}\,\underline{\Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow \blacktriangleleft t \blacktriangleright \prec t \succ e'}$$

Figure 8 Type Racket translation.

5.5 Behavioural Casts

A behavioural cast $\blacktriangleleft t \blacktriangleright a$ ensures that the result, a', will follow all of the type guarantees in t. The cast creates a wrapper which holds a reference to the original object a and dynamically enforces the type guarantees. The behavioural cast is generative as it creates both a class and a new instance of that class. The cast's dynamic semantics appear in Fig. 9. BC1 wraps a reference a with a freshly generated wrapper k for some type t, adds it into a new heap, and then hands it back to the top level dynamics. BC2 does the same, except that it wraps a with the guards to let a act like an instance of $\star$.

$$\text{BC1}\quad \frac{\begin{array}{c}\sigma(a) = C\{\overline{a_1}\} \qquad D \text{ fresh} \\ k = \mathsf{wrap}(C, \mathsf{getmds}(C, K), \mathsf{mtypes}(C, K), \mathsf{mtypes}(C', K), D) \qquad a'' \text{ fresh} \qquad \sigma' = \sigma[a'' \mapsto D\{a\}]\end{array}}{\mathsf{behcast}(a, C', \sigma, K) = K\,k\,a''\,\sigma'}$$

$$\text{BC2}\quad \frac{\begin{array}{c}\sigma(a) = C\{\overline{a_1}\} \\ D \text{ fresh} \qquad k = \mathsf{wrap}(C, \mathsf{getmds}(C, K), \mathsf{mtypes}(C, K), D) \qquad a' \text{ fresh} \qquad \sigma' = \sigma[a' \mapsto D\{a\}]\end{array}}{\mathsf{behcast}(a, \star, \sigma, K) = K\,k\,a'\,\sigma'}$$

Figure 9 Behavioural casts.

A wrapper generated for a type C is a subtype of C allowing us to refer to object instances and wrappers under the original, source language, types. Internally, wrappers either forward values, as in the case of getters and setters, or mimic the behaviour of the underlying object, in the case of methods, ensuring that the type guarantees of both the externally visible type and the wrapped object are respected. Wapper generation is implemented by wrap, a meta function of 5 arguments derived from the class table and the names of the types in question. An example is shown in next, where we wrap an instance of C, which we want to make sure acts like a D. wrap generates a proxy method for m, which acts like C's implementation of m, but has casts enforcing the type guarantees of D.

```
                                                         class D {
         class C {                class D {                that:C
wrap(      m (x :*):* {x}}  ,      m (x  A):B}     ) =     m(x:A):B {◀B▶ ≺B≻◀*▶ ≺*≻ x}
                                                         }
```

Copying the behaviour of the wrapped object is unusual, and is how we maintain one of the semantic Typed Racket. If we were to implement the wrappers directly, as seen in figure 10, we will diverge from Typed Racket despite remaining typesafe. The issue is that Typed Racket requires that self-references respect the external type invariants [13],

```
wrap( class C {                 ,  class D {              ) =
        m (x : *) : * { x }}         m (x : A) : B}

                          class D {
                            m(x : A) : B {
                              ◀B▶ ≺B≻ (<*>this.that())@m(◀*▶ ≺*≻ x) }
                          }
```

Figure 10 Naive wrapper generation

```
#lang typed/racket
(module untyped racket
  (define A% (class object% (super-new)
               (define/public (foo) "hello")
               (define/public (bar) (send this foo))))
  (define (gen) (make-object A%))
  (provide gen))

(define-type A (Class [foo (-> Integer)] [bar (-> Integer)]))
(require/typed 'untyped [gen (-> (Instance A))])
(send (gen) bar)
```

```
foo: broke its own contract
  promised: Integer
  produced: "hello"
  in: the foo method in
      the range of
      (->
       (object/c
        (bar (-> any/c Integer))
        (foo (-> any/c Integer))))
  contract from: (interface for gen)
  contract on: gen
  blaming: (interface for gen)
```

Figure 11 Typed Racket ensures that internal untyped calls respect external types

despite it not being strictly required for soundness, as seen in figure 11. We use the lifting mechanism, copying wrapped class functionality, to ensure that self-references refer to the wrapper, rather than to the wrapped object, thereby enforcing the dynamic type guarantees.

This issue is illustrated in figure 12, where the source program initially requires n to take A and return A, then requires it to take and return B. If we lift the methods up verbatim (adding only the trivial casts), then we end up with a type-incorrect wrapper. As a result, we have to dynamically insert casts inside of the lifted method bodies, ensuring that they remain type correct in the wrapper class. This operation is depicted in detail in the appendix.

```
class A {n(x:*):*{x}}
class B {m(x:*):*{x}}
class C {
  m(x:A):A {
    this.n(x) }
  n(x:A):A { x }}
class D {
  m(x:A):A { x }
  n(x:B):B { x }}
(<D>new C()).m(
  new C())
```

Source

```
class DW {
  that : C


  m(x:A):A {
    this.n(x) }

  n(x:B):B {
   ◀B▶ ≺B≻
    ◀A▶ ≺A≻ x }
}
```

Type-incorrect

```
class DW {
  that : C

  m(x:A):A {
    ◀A▶ ≺A≻ this.n(
      ◀B▶ ≺B≻ x) }

  n(x:B):B {
   ◀B▶ ≺B≻
      ◀A▶ ≺A≻ x }
}
```

Type-corrected

Figure 12 Wrapper generation

The other major concern when designing a wrapper-based protection system for objects is that losing methods is a real possibility. If a wrapper zealously enforces its type, then it will not wrap methods that do not appear on its type, which can then be lost to later untyped or more-typed code that is given that wrapper. Our approach avoids this by inserting "passthrough" methods that retain their original types and behaviors into the output wrappers, as illustrated in figure 13, where we make a C into a D and then back again. When we reduce the set of required methods by casting a C to a D, we retain the method m′ by adding it to the output class. Notably, this operation preserves existing subtyping relationships, as the generated wrappers only appear as values, and the operation only adds additional methods.

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

class CtoD {
    that: C
    m(x: ⋆): ⋆ {◀ ⋆ ▶≺ ⋆ ≻◀ E ▶≺ E ≻ x}
    m′(x: E): E {x}
}

class D {
    m(x: ⋆): ⋆ {x}
}

class CtoDtoC {
    that: CtoD
    m(x: E): E {◀ E ▶≺ E ≻◀ ⋆ ▶≺ ⋆ ≻
      ◀ E ▶≺ E ≻◀ ⋆ ▶≺ ⋆ ≻ x}
    m′(x: E): E {◀ E ▶≺ E ≻◀ E ▶≺ E ≻ x}
    m′(x: ⋆): ⋆ {◀ ⋆ ▶ this.m′(◀ E ▶≺ E ≻ x)}
}
```

Figure 13 Wrapper classes generated by ◀ C ▶ (◀ D ▶ **new** C())

The Typed Racket translation demonstrates some of the key issues inherent in a wrapper-based system, where wrapper-inserting casts build up very quickly internally, leading to the potential for a wrapper explosion, as previously noted by Takikawa et al [23], a point that is further highlighted by the number of casts inserted into the wrappers.

5.6 Monotonic Python

The Monotonic and Transient semantics both requires consistency. Consistency is the idea that two types do not contradict one another. For example, the types int and ⋆ are consistent, but the types int and string are not as no member of one could be in the other. The consistency operator (∼) in our formal system is implemented using the concept of meet described by the `tmeet` function. The `tmeet` function finds the common type between two given types, which if a common type exists, it implies that consistency holds between the two given types. Using the earlier example, the meet between int and ⋆ would be int, as int is the more restrictive of the two types. In effect, the `tmeet` function fulfills the same purpose as the ⊓ operator in [21].

$$\frac{\mathsf{tmeet}(t, t', \cdot, K) = t''\, K'}{K \vdash t \sim t'}\ \text{C1}$$

Consider the following example where we compute the meet of A and B, both classes have the same method, but with alternating the typing. In A, the return type is typed, whereas in B it is the argument that is typed. Calling the `tmeet` function on A and B will compute the new type C, whose method have is fully typed. The `tmeet` function will be described in more detail once the monotonic cast semantic is presented.

```
              class A {                           class A { ... }
                m(x: *): A {this}}                class B { ... }
tmeet(A, B, ·,                          ) = C,    class C {
              class B {                             m(x: B): A {this}
                m(x: B): * {this}}                }
```

Similar to the behavioural semantics, the monotonic semantics is implemented to the core of KafKa as a cast extension. For an expression e and type t, we write $\triangleleft\, \mathsf{t} \,\triangleright\, \mathsf{a}$ to depict the monotonic cast.

Static Typing

$$\textsc{w10}\quad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash \triangleleft\,\mathsf{t}\,\triangleright\,\mathsf{e} : \mathsf{t}}$$

Dynamics

$$\mathsf{K}\ \triangleleft\,\mathsf{t}\,\triangleright\,\mathsf{a}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{a}'\ \sigma' \qquad \mathsf{moncast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) = \mathsf{K}'\ \mathsf{a}'\sigma'$$

$$\mathcal{E} ::= \ldots \mid \triangleleft\,\mathsf{t}\,\triangleright\,\mathcal{E}$$

$$\textsc{pt}\quad \frac{\underline{\mathsf{K}} \vdash \underline{\mathsf{K}}\ \hookrightarrow_c\ \mathsf{K}' \qquad \underline{\mathsf{K}\ \cdot} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}}}{\underline{\mathsf{e}\ \mathsf{K}}\ \hookrightarrow_p\ \mathsf{e}'\ \mathsf{K}'}$$

$$\textsc{mct1}\quad \frac{\mathsf{D}\ \text{fresh} \qquad \mathsf{k} = \mathsf{monWrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{D}, \mathsf{K})}{\mathsf{mwrap}(\mathsf{C}, \mathsf{K}) = \mathsf{D}\ \mathsf{k}} \qquad \textsc{mct2}\quad \frac{}{\mathsf{mwrap}(\star, \mathsf{K}) = \star}$$

$$\textsc{cr1}\quad \frac{\overline{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{md}}\ \hookrightarrow_m\ \mathsf{md}'\ \mathsf{K}_m} \qquad \underline{\mathsf{K}} \vdash \underline{\mathsf{K}'}\ \hookrightarrow_c\ \mathsf{K}'' \qquad \overline{\mathsf{mwrap}(\mathsf{t}, \mathsf{K}'') = \mathsf{t}'\ \mathsf{K}'''}}{\underline{\mathsf{K}} \vdash \underline{\textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\}\ \mathsf{K}'}\ \hookrightarrow_c\ \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}'}\ \overline{\mathsf{md}'}\,\}\ \mathsf{K}''\ \mathsf{K}_m\ \overline{\mathsf{K}'''}} \qquad \textsc{cr2}\quad \frac{}{\underline{\mathsf{K}} \vdash \underline{\cdot}\ \hookrightarrow_c\ \cdot}$$

$$\textsc{mt}\quad \frac{\underline{\mathsf{K}\ \mathsf{this} : \mathsf{C}\ \mathsf{x} : \mathsf{t}} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}'} \hookrightarrow \mathsf{e}'\ \mathsf{K}_1 \qquad \mathsf{mwrap}(\mathsf{t}_1, \mathsf{K}) = \mathsf{t}_2\ \mathsf{K}_2 \qquad \mathsf{mwrap}(\mathsf{t}_1', \mathsf{K}) = \mathsf{t}_2'\ \mathsf{K}_3}{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{m}(\mathsf{x} : \mathsf{t}_1) : \mathsf{t}_1'\ \{\mathsf{e}\}}\ \hookrightarrow_m\ \mathsf{m}(\mathsf{x} : \mathsf{t}_2) : \mathsf{t}_2'\ \{\mathsf{e}'\}\ \mathsf{K}_1\ \mathsf{K}_2\ \mathsf{K}_3}$$

$$\textsc{mos1}\quad \frac{\mathsf{E}(\mathsf{x}) = \mathsf{t}}{\underline{\mathsf{K}\ \mathsf{E}} \vdash \underline{\mathsf{x}} \hookrightarrow \mathsf{x} \Uparrow \underline{\mathsf{t}}}$$

$$\textsc{mos2}\quad \frac{\begin{array}{cc}\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_3 \Uparrow \underline{\mathsf{C}}\ \mathsf{K}_1 & \mathsf{n}(\overline{\mathsf{t}_1}) : \mathsf{t}_2 \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \\ \mathsf{static}(\underline{\mathsf{t}_1}, \underline{\mathsf{K}}, \cdot) \vee \mathsf{n} = \mathsf{f} & \overline{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\mathsf{t}_1} \hookrightarrow \mathsf{e}_4\ \mathsf{K}_2}\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{n}(\overline{\mathsf{e}_2})} \hookrightarrow \mathsf{e}_3.\mathsf{n}(\overline{\mathsf{e}_4}) \Uparrow \underline{\mathsf{t}_2}\ \mathsf{K}_1\ \overline{\mathsf{K}_2}}$$

$$\textsc{mos3}\quad \frac{\begin{array}{cc}\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \hookrightarrow \mathsf{e}_3 \Uparrow \underline{\mathsf{C}}\ \mathsf{K}_1 & \mathsf{m}(\mathsf{t}_1) : \mathsf{t}_2 \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \\ \neg\mathsf{static}(\mathsf{t}_1, \mathsf{K}, \cdot) & \overline{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\mathsf{t}_1} \hookrightarrow \mathsf{e}_4\ \mathsf{K}_2}\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2)} \hookrightarrow (\triangleleft \star \triangleright \mathsf{e}_3)@\mathsf{m}(\triangleleft \star \triangleright \mathsf{e}_4) \Uparrow \underline{\mathsf{t}_2}\ \mathsf{K}_1\ \overline{\mathsf{K}_2}}$$

$$\textsc{mos4}\quad \frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}_3\ \mathsf{K}_1 \qquad \underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_2} \Downarrow \underline{\star} \hookrightarrow \mathsf{e}_4\ \mathsf{K}_2}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}_1@\mathsf{m}(\mathsf{e}_2)} \hookrightarrow \mathsf{e}_3@\mathsf{m}(\mathsf{e}_4) \Uparrow \underline{\star}\ \mathsf{K}_1\ \mathsf{K}_2}$$

$$\textsc{mos5}\quad \frac{\begin{array}{c}\overline{\underline{\mathsf{K}\ \mathsf{E}} \vdash \underline{\mathsf{e}_1} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}_2\ \mathsf{K}} \qquad \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \\ \mathsf{D}\ \text{fresh} \qquad \mathsf{k} = \mathsf{monWrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{D}, \mathsf{K})\end{array}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\textbf{new}\ \mathsf{C}(\overline{\mathsf{e}_1})} \hookrightarrow \textbf{new}\ \mathsf{D}(\textbf{new}\ \mathsf{C}(\overline{\mathsf{e}_2})) \Uparrow \underline{\mathsf{C}}\ \mathsf{K}\ \mathsf{k}}$$

$$\textsc{moa1}\quad \frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}'}\ \mathsf{K} \qquad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'\ \mathsf{K}} \qquad \textsc{moa2}\quad \frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}'}\ \mathsf{K} \qquad \mathsf{K} \vdash \mathsf{t}\ \sim\ \mathsf{t}'}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \triangleleft\,\mathsf{t}\,\triangleright\,\mathsf{e}'\ \mathsf{K}}$$

Figure 14 Monotonic translation for bidirectional expressions

The translation from the Monotonic surface language is in Fig. 14. The rule PT translates programs, CR1 and CR2 translates classes, and MT translates methods. In the MT and CR1 rules, we are actively replacing every type the user wrote with ones generated by the monWrap function. The details of the monWrap function will be discussed later, but for the purposes of the translation, we can assume that monWrap replaces every typed function with an untyped one unless there are no $\star$ types transitively contained within the function's argument. The `static` function states whether a type contains only typed methods. The rule MOS2 translates typed methods, MOS2 translate methods that are not purely typed. MOS4 translate untyped methods, MOS2 translate object creation. The purpose of the monWrap function is further evidenced as the handling of calls to methods need to appropriately casted if the typing nature is unclear. Monotonic has identical analytical cast insertion to the transient system, combining subtyping with consistency, which allow use the of either subtyping or consistency in type coercions

5.7 Monotonic Casts

The monotonic cast $\triangleleft$C$\triangleright$a imposes the type C onto the object at a and every object transitively reachable from a. The motivation for this cast comes from maintaining the type correctness of every reference in the program [21].

```
class C { f : * }
class D {
  m (x:*):* {x}
}

d:* = new D()
a:* = new C(d)
[heap snapshot 1]

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

e:E = <E>a
[heap snapshot 2]
d.m(d)
```

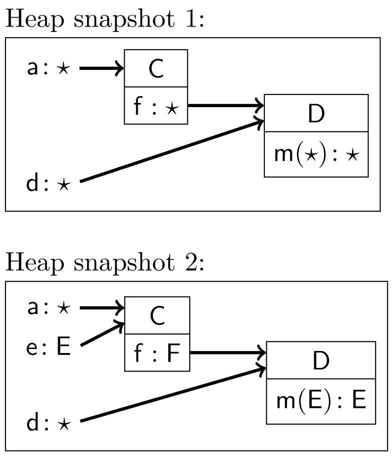


Figure 15 Execution under the monotonic semantics

Figure 15 presents an example of the monotonic cast. A new instance of class D is created, then an instance of class C is created pointing to it. As shown by heap snapshot 1, the field f in C has type $\star$, and the method m in D has dynamic argument and return type. Classes E and F are introduced and they are structurally identical to C and D, but with specific types for their fields and methods. When the instance of class C referenced by a is casted to E, the monotonic semantics will ensure any access to field f or method m from anywhere in the program will follow the type invariants asserted by class E.

However, as shown in heap snapshot 2, the reference d still refer to the instance of D, from earlier, and d does not know the existence of the new invariant applied to class D. Despite being typed $\star$, when d try to invoke method m on the apparently dynamic object, a cast failure will occur. The monotonic semantics ensures that the provided argument must be of type E.

This example highlights the need for the monotonic cast semantics to be generative. In order to guarantee the type correctness of all references, monotonic has to generate guards that check the argument and return type of every method, and the type of any field set, ensuring that they maintain consistency with the maximally static type. Additionally, monotonic ensures that prior references respect future type invariants by performing an in-place update of any object whose type is altered by the cast.

In the extreme case, a single cast would retype every object in the heap. In order to prevent the pathological case, the monotonic casts only modify the heap if the target C has fewer occurrences of $\star$ than the original type of the object at a (and transitively so for types appearing in t and reachable from a). When a type C does not have any occurrence of the dynamic type, denoted by static(C,K,V), the monotonic casts will leave the object and the heap unchanged.

The monotonic cast semantics in KafKa cannot be implemented using the in-place replacement technique[21]. In order to ensure the type of a field is respected, the monotonic cast will insert a setter method checking the value given to it against the type of the field, when a lesser-typed call site is used to set the field. However, KafKa prohibits the explicit existence of a field and its getter and setter methods. As a field and its getter and setter method must be in-sync at all times. This means the monotonic cast semantics is implemented in KafKa by adding a single wrapper over each object, similar to the "identity preserving membrane" described in [24]. The wrappers will preserve the externally-facing interface (up to a point) and the relations between classes, while ensuring all properties in the monotonic system are guaranteed.

As a result of generating the wrappers (and the static translation), the simple example shown in figure 15 does not correctly exhibit the correct behavior of the monotonic semantics in KafKa, instead the example presented in figure 16 depicts the correct semantics for KafKa's monotonic cast. For compactness, the class definitions in figure 15, and the class definitions that will be generated but not appear in the dynamic execution have been elided.

Figure 16 aims to illustrate the nature of the monotonic wrappers. In this example, the state of the heap is highlighted at two discrete times, before and after the cast of the reference a to the type E. The wrapper CW and DW is associated with enforcing the class C and D, respectively, and wraps over an underlying object of known type, enforcing its type guarantees.

The monotonic semantics will lift the method bodies into wrapper classes, as seen in the Typed Racket semantics but for different reasons. While Typed Racket needs to enforce the observed type on an underlying untyped object. The monotonic semantics need the combination of the wrapper and underlying object to act as if they are exactly the monotonically specified type. The result of this can be seen in the translation above for m.

A notable behavior of the monotonic semantics illustrated in figure 16 is the erasure on every call site for m, with the exception of the final definition in FDW. Under the monotonic semantics, every call site with some untyped component will be adjusted further, ensuring any partially typed call may be invalidated in the future. As a consequence, every call, even to a superficially typed call site (one with an argument type that may contain a $\star$ deeply embedded in the type), must be treated as an untyped call site.

In the monotonic semantics, if we encounter a cast on a reference, that cast needs to be enforced upon that reference for the rest of its lifetime. KafKa's monotonic cast semantics handles this by taking the previously-existing type on the object, computing the meet of that old type and the new type, producing a new wrapper for the object that replaces the previous wrapper. The wrappers ECW and FDW are created when a is casted to E. To ensure

Translated Program:

```
d:* = new DW(new D())
a:* = new CW(new C(d))
[heap snapshot 1]
e:E = <E>a
[heap snapshot 2]
d.m(d)
```

Class C Wrappers:

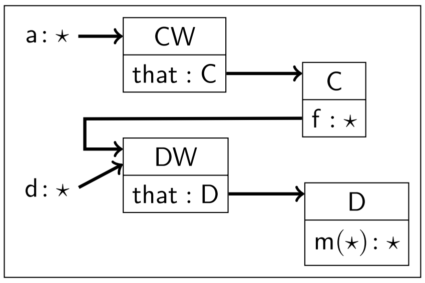
```
class CW {
  that : C
  f():* { <*>this.that().f() }
  f(x:*):* {
    <*>this.that().f(<*>x)}
}
class ECW {
  that : C
  f():* { <*><FW>this.that.f() }
  f(x:*):* {
    <*><FW>this.that().f(<*><FW>x)}
}
```

Class D Wrappers:

```
class DW {
  that : D
  m (x:*):* { <*><*>x }
}


class FDW {
  that : D
  m(x:*):* { <*><EW><*><EW>x }
  m(x:EW):EW { <EW><*><EW>x }
}
```

Heap snapshot 1:



Heap snapshot 2:

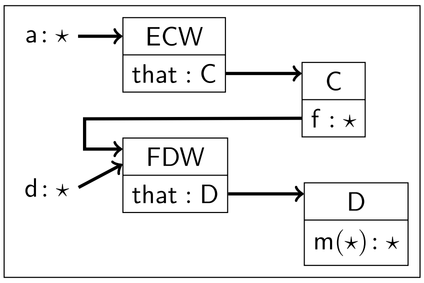


Figure 16 Execution under the monotonic semantics

that types for fields are respected, we take a conservative approach, as seen in ECW. When a field on an underlying object is set, we first cast it to the required type (in this case FW). Then a further $\star$ cast is applied to the type of the underlying storage cell. For retrieving values, we reverse this process.

The result wrappers are sometimes overly conservative. From the semantics of the meet operation, we know that the first cast to the protected type is sufficient to ensure compatibility with the underlying memory cell type, and we also know that casting to the required type before returning is ignoring the heap consistency invariant that monotonic provides, allowing for casts on retrieval to be ignored. However, the KafKa type system does not allow us to make either of these shortcuts while still retaining semantic correctness, and as a result, we are forced to double up on the casts over the getter and setter methods, as seen in ECW. Another point of interest is the alteration happening in FDW. When the class D is specialized to F, several additional casts are inserted, as well as a wholly new call site for m. This new call site is fully typed, despite previously stating that we could not add any typed call sites as they might be changed in the future. In this case, we are able to insert the additional static call site because we know that E will not be able to be further altered under the monotonic semantics, as E and every type referenced from it have no instances of the type $\star$ in them, which might be altered in the future by another monotonic cast.

The monotonic cast semantics is formally defined by the moncast rule, which takes the address of the cast object a, a target type t, the current heap σ, and class table K, and gives

$$\textsc{cm}\quad \frac{\mathsf{retype}(\mathsf{a},\mathsf{t},\cdot,\sigma,\mathsf{K}) = \tau\ \mathsf{K}' \qquad \mathsf{spec}(\mathsf{dom}(\tau),\tau,\sigma,\mathsf{K}') = \sigma'\ \mathsf{K}''}{\mathsf{moncast}(\mathsf{a},\mathsf{t},\sigma,\mathsf{K}) = \mathsf{K}''\ \sigma'}$$

Figure 17 Monotonic cast semantics

an update heap $\sigma'$and an updated class table K'.

The functionality of the moncast rule is broken into two functions. The first function (retype) identities the correct target type for every object affect by the monotonic cast, storing the gathered objects and their types in the meta-variable τ, a mapping from addresses to classes, creating a blueprint for the sub-heap affected by the monotonic cast. The second function (spec) updates the heap and class table with the objects and types listed in τ.

$$\boxed{\mathsf{tmeet}(\mathsf{t},\mathsf{t}',\mathsf{P},\mathsf{K}) = \mathsf{t}''\ \mathsf{K}'}$$

$$\mathsf{P} ::= \cdot \mid \mathsf{P}[(\mathsf{C},\mathsf{D}) \mapsto \mathsf{E}]$$

$$\textsc{tm1}\quad \frac{}{\mathsf{tmeet}(\mathsf{C},\star,\mathsf{P},\mathsf{K}) = \mathsf{C}\,\mathsf{K}} \qquad \textsc{tm2}\quad \frac{}{\mathsf{tmeet}(\star,\mathsf{C},\mathsf{P},\mathsf{K}) = \mathsf{C}\,\mathsf{K}} \qquad \textsc{tm3}\quad \frac{}{\mathsf{tmeet}(\mathsf{t},\mathsf{t},\mathsf{P},\mathsf{K}) = \mathsf{t}\,\mathsf{K}}$$

$$\textsc{tm4}\quad \frac{\begin{array}{c}\mathsf{E}\ \text{fresh} \qquad (\mathsf{C},\mathsf{D}) \notin \mathsf{P} \qquad \mathsf{P}' = \mathsf{P}[(\mathsf{C},\mathsf{D}) \mapsto \mathsf{E}] \qquad \mathsf{mtypes}(\mathsf{C},\mathsf{K}) = \overline{\mathsf{mt}} \\ \mathsf{mtypes}(\mathsf{D},\mathsf{K}) = \overline{\mathsf{mt}'} \qquad \mathsf{mmeet}(\overline{\mathsf{mt}},\overline{\mathsf{mt}'},\mathsf{P}',\mathsf{K}) = \overline{\mathsf{mt}''}\,\mathsf{K}' \qquad \mathsf{K}'' = \mathsf{K}'\ \mathsf{typegen}(\overline{\mathsf{mt}''},\mathsf{E})\end{array}}{\mathsf{tmeet}(\mathsf{C},\mathsf{D},\mathsf{P},\mathsf{K}) = \mathsf{E}\,\mathsf{K}''}$$

$$\textsc{tm5}\quad \frac{\mathsf{P}(\mathsf{C},\mathsf{D}) = \mathsf{E}}{\mathsf{tmeet}(\mathsf{C},\mathsf{D},\mathsf{P},\mathsf{K}) = \mathsf{E}\,\mathsf{K}}$$

Figure 18 The `tmeet` function

The retype function utilizes the meet operation in determining the correct casting type for every object transitively reachable from the cast object. Figure 18 presents the tmeet function, which computes the meet between two types. The tmeet function takes four arguments, the two types being meet, a meta-variable P, and the class table K. The meta-variable P denotes a list of mappings from a pair of types to the type produced from their meet. The rules TM1 and TM2 describe the meet of a $\star$. The rule TM3 describes the identity meet. The rule TM5 describes the recursive case of our meet operation. The rule TM4 retrieves the method typing for each type, computes the meet of those method typing (by calling an auxiliary function (mmeet), and updates the class table with a new class containing the method typing produced from the meet.

The meta-variable P becomes crucial when considering the example in figure 19, where the meet of A and B recursively depends on the meet of A and B. If we proceeded naively, without P, TM4 would invoke TM4 again to compute the meet of the arguments to m. We use P and TM5 to avoid this.

TM4 invokes mmeet with an enriched P, in much the same manner that the subtyping relation does, inserting the future result of the meet of A and B (E in this example) into P,

```
                class A {                         class A { ... }
                   m(x: A): A {this}}             class B { ... }
tmeet(A, B, ·,                          ) = E,    class E {
                class B {                           m(x: E): E {this}
                   m(x: B): B {this}}             }
```

Figure 19 `tmeet` example over recursive classes

will then be used by TM5 to terminate the recursion.

6 Conclusion

Gradual typing is no longer simply a popular research topic in academia. Real-world applications are being written with gradual types and the tug of war between soundness and performance is being played out in multiple language designs. It is the responsibility of language researchers to present a clear understanding of each viable gradual typing idiom available. We have introduced KafKa, a formal language that serves as the foundation for deconstructing five existing gradual typing systems: Typescript, Thorn, Transient Python, Type Racket, and Monotonic Python. KafKa offers the opportunity for comparing and contrasting these gradual typing systems within an unified framework. The translations to KafKa for each gradual typing system highlights the essences which makes each system unique.

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 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`.

18 Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004.

19 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`.

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, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. *Monotonic References for Efficient Gradual Typing*, pages 432–456. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. URL: `http://dx.doi.org/10.1007/978-3-662-46669-8_18`, `doi:10.1007/978-3-662-46669-8_18`.

22 Benjamin C. Pierce and David N. Turner. Local type inference. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265, New York, NY, 1998. URL: `citeseer.nj.nec.com/pierce98local.html`.

23 Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In *???*, volume 37, 2015.

24 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent Object Proxies for JavaScript (Artifact). *Dagstuhl Artifacts Series*, 1(1):1–2, 2015. URL: `http://drops.dagstuhl.de/opus/volltexte/2015/5511`, `doi:http://dx.doi.org/10.4230/DARTS.1.1.2`.

A Auxiliary Definitions for KafKa

A.1 Syntax

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

Figure 20 KafKa Syntax.

A.2 Semantics

$\boxed{\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'}$ e σ evaluates to e$'$ in a step

$$
\begin{array}{lll}
\mathsf{K}\ \mathbf{new}\ \mathsf{C}(\overline{\mathsf{a}})\ \sigma & \to \mathsf{K}\ \mathsf{a}'\ \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\overline{\mathsf{a}}\}] & \mathsf{a}'\ \mathsf{fresh} \\
\mathsf{K}\ \mathsf{a.f()}\ \sigma & \to \mathsf{K}\ [\mathsf{a/this}]\mathsf{e}\ \sigma & \mathsf{f()\colon t}\ \{\mathsf{e}\} \in \mathsf{K(C)} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a.f(a')}\ \sigma & \to \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a'/x}]\mathsf{e}\ \sigma & \mathsf{f(x\colon t)\colon t}\ \{\mathsf{e}\} \in \mathsf{K(C)} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a.f()}\ \sigma & \to \mathsf{K}\ \mathsf{a}'\ \sigma & \mathsf{read}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{K}) = \mathsf{a}' \\
\mathsf{K}\ \mathsf{a.f(a')}\ \sigma & \to \mathsf{K}\ \mathsf{a}'\ \sigma' & \mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{a}', \mathsf{K}) = \sigma' \\
\mathsf{K}\ \mathsf{a.m}_{\mathsf{t}'\to\mathsf{t}}(\mathsf{a}')\ \sigma & \to \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a'/x}]\mathsf{e}\ \sigma & \mathsf{m(x\colon t')\colon t}\ \{\mathsf{e}\} \in \mathsf{K(C)} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \\
\mathsf{K}\ \mathsf{a@m(a')}\ \sigma & \to \mathsf{K}\ [\mathsf{a/this}\ \mathsf{a'/x}]\mathsf{e}\ \sigma & \mathsf{m(x\colon \star)\colon \star}\ \{\mathsf{e}\} \in \mathsf{K(C)} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \\
\mathsf{K}\ <\star>\mathsf{a}\ \sigma & \to \mathsf{K}\ \mathsf{a}\ \sigma & \\
\mathsf{K}\ <\mathsf{D}>\mathsf{a}\ \sigma & \to \mathsf{K}\ \mathsf{a}\ \sigma & \mathsf{K} \vdash \mathsf{C} <: \mathsf{D} \wedge \sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}}\} \\
\mathsf{K}\ \mathcal{E}[\mathsf{e}]\ \sigma & \to \mathsf{K}'\ \mathcal{E}[\mathsf{e}']\ \sigma' & \mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'
\end{array}
$$

$$
\begin{array}{llllllll}
\mathcal{E} ::= \mathcal{E}\mathsf{.f()} & \mid\ \mathcal{E}\mathsf{.f(e)} & \mid\ \mathsf{a.f}(\mathcal{E}) & \mid\ \mathcal{E}\mathsf{.m}_{\mathsf{t}\to\mathsf{t}}(\mathsf{e}) & \mid\ \mathsf{a.m}_{\mathsf{t}\to\mathsf{t}}(\mathcal{E}) & \mid\ \mathcal{E}\mathsf{@m(e)} & \mid\ \mathsf{a@m}(\mathcal{E}) \\
\quad \mid\ <\mathsf{t}>\mathcal{E} & \mid\ \mathbf{new}\ \mathsf{C}(\overline{\mathsf{a}}\,\mathcal{E}\,\overline{\mathsf{e}}) & \mid\ \square & & & &
\end{array}
$$

Figure 21 KafKa Semantics.

A.3 Subtyping

The structural subtype relation, written $\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$, asserts that t is a subtype of t' in the environment μ composing a set of subtype relations and a class table K. The set of subtype relations can be omitted if its empty.

$\boxed{\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'}$ t is a subtype of t'

$$\textsc{SRef}\ \frac{}{\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}} \qquad \textsc{SAss}\ \frac{\mathsf{C} <: \mathsf{D} \in \mu}{\mu\ \mathsf{K} \vdash \mathsf{C} <: \mathsf{D}}$$

$$\textsc{SRec}\ \frac{\mu' = \mu\ \mathsf{C} <: \mathsf{D} \qquad \mathsf{mt} \in \mathsf{mtypes}(\mathsf{D}, \mathsf{K}) \implies \mathsf{mt}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})\ .\ \mu'\ \mathsf{K} \vdash \mathsf{mt} <: \mathsf{mt}'}{\mu\ \mathsf{K} \vdash \mathsf{C} <: \mathsf{D}}$$

$\boxed{\mu\ \mathsf{K} \vdash \mathsf{mt} <: \mathsf{mt}'}$ mt is a subtype of mt'

$$\textsc{SMet}\ \frac{\mu\ \mathsf{K} \vdash \mathsf{t}_1 <: \mathsf{t}_2 \qquad \mu\ \mathsf{K} \vdash \mathsf{t}'_2 <: \mathsf{t}'_1}{\mu\ \mathsf{K} \vdash \mathsf{m}(\mathsf{t}_1) : \mathsf{t}'_1 <: \mathsf{m}(\mathsf{t}_2) : \mathsf{t}'_2} \qquad \textsc{SGet}\ \frac{\mu\ \mathsf{K} \vdash \mathsf{t}_1 <: \mathsf{t}_2}{\mu\ \mathsf{K} \vdash \mathsf{f}() : \mathsf{t}_1 <: \mathsf{f}() : \mathsf{t}_2} \qquad \textsc{SSet}\ \frac{\mu\ \mathsf{K} \vdash \mathsf{t}_1 <: \mathsf{t}_2}{\mu\ \mathsf{K} \vdash \mathsf{f}(\mathsf{t}_1) : \mathsf{t}_1 <: \mathsf{f}(\mathsf{t}_2) : \mathsf{t}_2}$$

A.4 Well-formedness

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

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

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

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

$$\textsc{WC}\ \frac{\mathsf{overloading}_{\emptyset}(\overline{\mathsf{md}}\ \overline{\mathsf{fd}})\ \checkmark \qquad \mathsf{fd} \in \overline{\mathsf{fd}} \implies \mathsf{K} \vdash \mathsf{fd}\ \checkmark \qquad \mathsf{md} \in \overline{\mathsf{md}} \implies \mathsf{this} : \mathsf{C}\ \ \sigma\, \mathsf{K} \vdash \mathsf{md}\ \checkmark}{\sigma\, \mathsf{K} \vdash \textbf{class}\ \mathsf{C}\, \{\, \overline{\mathsf{fd}}\ \overline{\mathsf{md}}\, \}\ \checkmark}$$

The overloading auxiliary function states that there are no overloaded field or method names within the given field and method definitions.

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

$$\frac{\Gamma\ \mathsf{x}\colon\star\ \sigma\,\mathsf{K}\vdash\mathsf{e}:\star\qquad \mathsf{K}\vdash\star\ \checkmark}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{m}(\mathsf{x}\colon\star)\colon\star\ \{\mathsf{e}\}\ \checkmark}\ \text{WT1}\qquad\qquad \frac{\Gamma\ \mathsf{x}\colon\star\ \sigma\,\mathsf{K}\vdash\mathsf{e}:\star\qquad \mathsf{K}\vdash\star\ \checkmark}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{f}(\mathsf{x}\colon\star)\colon\star\ \{\mathsf{e}\}\ \checkmark}\ \text{WS1}$$

$$\frac{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{e}:\star\qquad \mathsf{K}\vdash\star\ \checkmark}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{f}()\colon\star\ \{\mathsf{e}\}\ \checkmark}\ \text{WG1}$$

$$\frac{\begin{array}{c}\Gamma\ \mathsf{x}\colon\mathsf{C}\ \ \sigma\,\mathsf{K}\vdash\mathsf{e}:\mathsf{C}'\\ \mathsf{K}\vdash\mathsf{C}\ \checkmark\qquad \mathsf{K}\vdash\mathsf{C}'\ \checkmark\end{array}}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{m}(\mathsf{x}\colon\mathsf{C})\colon\mathsf{C}'\ \{\mathsf{e}\}\ \checkmark}\ \text{WT2}\qquad\qquad \frac{\Gamma\ \mathsf{x}\colon\mathsf{C}'\ \ \sigma\,\mathsf{K}\vdash\mathsf{e}:\mathsf{C}\qquad \mathsf{K}\vdash\mathsf{C}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{f}(\mathsf{x}\colon\mathsf{C})\colon\mathsf{C}\ \{\mathsf{e}\}\ \checkmark}\ \text{WS2}$$

$$\frac{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{e}:\mathsf{C}\qquad \mathsf{K}\vdash\mathsf{C}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K}\vdash\mathsf{f}()\colon\mathsf{C}\ \{\mathsf{e}\}\ \checkmark}\ \text{WG2}$$

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

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

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

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

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

$$\frac{\begin{array}{c}\mathsf{a}'\mapsto\mathsf{C}\{\mathsf{a}_1\ldots\mathsf{a}_n\}\ \in\ \sigma\ \Longrightarrow\ \textbf{class}\ \mathsf{C}\,\{\,\mathsf{f}_1:\mathsf{t}_1\ldots\mathsf{f}_n:\mathsf{t}_n\ \overline{\mathsf{md}}\,\}\in\mathsf{K}\ \wedge\\ \cdot\,\sigma\,\mathsf{K}\vdash\mathsf{a}_1:\mathsf{t}_1\ \ldots\ \cdot\,\sigma\,\mathsf{K}\vdash\mathsf{a}_n:\mathsf{t}_n\end{array}}{\mathsf{K}\vdash\sigma\ \checkmark}\ \text{WH}$$

A.5 Expression typing

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

Field access rules W3 and W4 require a typed receiver, since $\star$ does not have any methods a receiver typed at $\star$ will never typecheck.

Shallow casts, W9, do not change the type of the expression, as we are casting to the name of t not to t.

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

$$\frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{x} : \mathsf{t}}\ \text{W1} \qquad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}' \qquad \cdot\,\mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}}\ \text{W2}$$

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C} \qquad \mathsf{f}() : \mathsf{t} \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.f}() : \mathsf{t}}\ \text{W3} \qquad \frac{\begin{array}{c}\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C} \\ \mathsf{f}(\mathsf{t}) : \mathsf{t} \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t}\end{array}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.f}(\mathsf{e}') : \mathsf{t}}\ \text{W4}$$

$$\frac{\begin{array}{c}\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{C} \\ \mathsf{m}(\mathsf{t}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t}\end{array}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e.m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}') : \mathsf{t}'}\ \text{W5} \qquad \frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star \qquad \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \star}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e@m}(\mathsf{e}') : \star}\ \text{W6}$$

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}_1 : \mathsf{t}_1 \ldots \Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e}_n : \mathsf{t}_n \qquad \overline{\mathsf{fd}} = \mathsf{f}_1 : \mathsf{t}_1 \ldots \mathsf{f}_n : \mathsf{t}_n \qquad \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K}}{\Gamma\,\sigma\,\mathsf{K} \vdash \textbf{new}\ \mathsf{C}(\mathsf{e}_1 \ldots \mathsf{e}_n) : \mathsf{C}}\ \text{W7}$$

$$\frac{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash <\mathsf{t}> \mathsf{e} : \mathsf{t}}\ \text{W8} \qquad \frac{\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}'}\}}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}}\ \text{W9} \qquad \frac{}{\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \star}\ \text{W10}$$

A.6 Field read

The function $\mathsf{read}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{K})$ reads the value in the field f in the object a.

$$\mathsf{read}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{K}) = \mathsf{a}' \quad \textit{if} \quad \begin{cases}\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1 \ldots \mathsf{a}_n \mathsf{a}' \ldots\} \\ \textbf{class}\ \mathsf{C}\,\{\,\mathsf{f}_1 : \mathsf{t}_1 \ldots \mathsf{f}_n : \mathsf{t}_n \mathsf{f} : \mathsf{t} \ldots\ \overline{\mathsf{md}}\,\} \in \mathsf{K}\end{cases}$$

A.7 Field write

The function $\mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{a}', \mathsf{K})$ denotes writing of value a′ to the field f of the object stored at a in σ.

$$\mathsf{write}(\sigma, \mathsf{a}, \mathsf{f}, \mathsf{a}', \mathsf{K}) = \sigma[\mathsf{a} \mapsto \mathsf{C}\{\mathsf{a}_1 \ldots \mathsf{a}_n\ \mathsf{a}' \ldots\}] \ \textit{if} \ \begin{cases}\sigma(\mathsf{a}) = \mathsf{C}\{\mathsf{a}_1 \ldots \mathsf{a}_n\ \mathsf{a}'' \ldots\} \\ \textbf{class}\ \mathsf{C}\,\{\,\mathsf{f}_1 : \mathsf{t}_1 \ldots \mathsf{f}_n : \mathsf{t}_n\ \mathsf{f} : \mathsf{t} \ldots\ \overline{\mathsf{md}}\,\} \in \mathsf{K}\end{cases}$$

A.8 Method overloading

The function $\mathsf{overloading}(\overline{\mathsf{md}}\ \overline{\mathsf{fd}})$ there are no overloaded method names in the given method definitions.

$\boxed{\mathsf{overloading}(\overline{\mathsf{md}}\ \overline{\mathsf{fd}})}$ Method overloading

$$\textsc{mo1}\quad \frac{}{\mathsf{overloading}_{\mathsf{MD}}(\emptyset)} \qquad \textsc{mo2}\quad \frac{\mathsf{m}(\mathsf{t}) : \mathsf{t} \notin \mathsf{MD} \qquad \mathsf{MD}' = \mathsf{MD}, \mathsf{m}(\mathsf{t}) : \mathsf{t} \qquad \mathsf{overloading}_{\mathsf{MD}'}(\overline{\mathsf{md}})}{\mathsf{overloading}_{\mathsf{MD}}(\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\mathsf{e}\}\ \overline{\mathsf{md}}\ \overline{\mathsf{fd}})}$$

$$\textsc{mo3}\quad \frac{\mathsf{f}(\mathsf{t}) : \mathsf{t} \notin \mathsf{MD} \qquad \mathsf{MD}' = \mathsf{MD}, \mathsf{f}(\mathsf{t}) : \mathsf{t} \qquad \mathsf{overloading}_{\mathsf{MD}'}(\overline{\mathsf{md}}, \overline{\mathsf{fd}})}{\mathsf{overloading}_{\mathsf{MD}}(\mathsf{f}(\mathsf{x} : \mathsf{t}) : \mathsf{t}\ \{\mathsf{e}\}\ \overline{\mathsf{md}}\ \overline{\mathsf{fd}})}$$

$$\textsc{mo4}\quad \frac{\mathsf{f}() : \mathsf{t} \notin \mathsf{MD} \qquad \mathsf{MD}' = \mathsf{MD}, \mathsf{f}() : \mathsf{t} \qquad \mathsf{overloading}_{\mathsf{MD}'}(\overline{\mathsf{md}}, \overline{\mathsf{fd}})}{\mathsf{overloading}_{\mathsf{MD}}(\mathsf{f}() : \mathsf{t}\ \{\mathsf{e}\}\ \overline{\mathsf{md}}\ \overline{\mathsf{fd}})}$$

$\boxed{\mathsf{overloading}(\overline{\mathsf{fd}})}$ field overloading

$$\textsc{fo1}\quad \frac{\mathsf{f} \notin \mathsf{names}(\mathsf{MD}) \qquad \mathsf{MD}' = \mathsf{MD}, \mathsf{f} : \mathsf{t} \qquad \mathsf{overloading}_{\mathsf{MD}'}(\overline{\mathsf{fd}})}{\mathsf{overloading}_{\mathsf{MD}}(\mathsf{f} : \mathsf{t}\ \overline{\mathsf{fd}})}$$

B Complete translation for TypeScript

B.1 TypeScript translation for program, class, and method

$\boxed{\underline{\mathsf{e}\ \mathsf{K}} \ \hookrightarrow_p\ \mathsf{e}'\ \mathsf{K}'}$ TypeScript translation for programs

$\boxed{\underline{\mathsf{K}} \vdash \underline{\textbf{class}\ \mathsf{C}\,\{\ldots\ \ldots\}\ \mathsf{K}} \ \hookrightarrow_c\ \textbf{class}\ \mathsf{C}\,\{\ldots\ \ldots\}\ \mathsf{K}}$ TypeScript translation for classes

$\boxed{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{md}} \ \hookrightarrow_m\ \mathsf{md}'\ \mathsf{K}}$ TypeScript translation for methods

$$\textsc{pt}\quad \frac{\underline{\mathsf{K}} \vdash \underline{\mathsf{K}} \ \hookrightarrow_c\ \mathsf{K}' \qquad \underline{\mathsf{K}\ \cdot} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}}}{\underline{\mathsf{e}\ \mathsf{K}} \ \hookrightarrow_p\ \mathsf{e}'\ \mathsf{K}'}$$

$$\textsc{cr1}\quad \frac{\overline{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{md}} \ \hookrightarrow_m\ \mathsf{md}'} \qquad \underline{\mathsf{K}} \vdash \underline{\mathsf{K}'} \ \hookrightarrow_c\ \mathsf{K}''}{\underline{\mathsf{K}} \vdash \underline{\textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\}\ \mathsf{K}'} \ \hookrightarrow_c\ \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \star}\ \overline{\mathsf{md}'}\,\}\ \mathsf{K}''} \qquad \textsc{cr2}\quad \frac{}{\underline{\mathsf{K}} \vdash \underline{\cdot} \ \hookrightarrow_c\ \cdot}$$

$$\textsc{mt}\quad \frac{\underline{\mathsf{K}\ \mathsf{this} : \mathsf{C}\ \mathsf{x} : \mathsf{t}} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}'} \hookrightarrow \mathsf{e}'}{\underline{\mathsf{K}\ \mathsf{C}} \vdash \underline{\mathsf{m}(\mathsf{x} : \mathsf{t}) : \mathsf{t}'\ \{\mathsf{e}\}} \ \hookrightarrow_m\ \mathsf{m}(\mathsf{x} : \star) : \star\ \{<\star>\ \mathsf{e}'\}}$$

B.2 TypeScript synthetic translation for expressions

$\boxed{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}}}$ e translates to e' producing type t, with context Γ and class table K

TSS1
$$\frac{\Gamma(x) = t}{\underline{K\ \Gamma} \vdash \underline{x} \hookrightarrow x \Uparrow \underline{t}}$$

TSS2
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \qquad m(\overline{t_1}) : t_2 \in \mathrm{mtypes}(C, K) \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow^{<\star>} e_3@m(<\star>\ e_4) \Uparrow \underline{t_2}}$$

TSS3
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{\star} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{\star} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m(e_4) \Uparrow \underline{\star}}$$

TSS4
$$\frac{E(\mathrm{this}) = C \qquad f(\overline{t_1}) : t_2 \in \mathrm{mtypes}(C, K) \qquad \underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t_1} \hookrightarrow e'}{\underline{K\ \Gamma} \vdash \underline{\mathrm{this}.f(e)} \hookrightarrow \mathrm{this}.f(e') \Uparrow \underline{\star}}$$

TSS5
$$\frac{E(\mathrm{this}) = C \qquad f() : t_2 \in \mathrm{mtypes}(C, K)}{\underline{K\ \Gamma} \vdash \underline{\mathrm{this}.f()} \hookrightarrow \mathrm{this}.f() \Uparrow \underline{\star}}$$

TSS6
$$\frac{\overline{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{t} \hookrightarrow e_2} \qquad \textbf{class}\ C\,\{\,\overline{f : t}\ \overline{md}\,\}}{\underline{K\ \Gamma} \vdash \underline{\textbf{new}\ C(\overline{e_1})} \hookrightarrow^{<\star>} \textbf{new}\ C(\overline{e_2}) \Uparrow \underline{C}}$$

B.3 TypeScript analytic translation for expressions

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$ e translates to e' against type t, with context Γ and class table K

TSA1
$$\frac{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash t' <: t}{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$$

TSA2
$$\frac{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{\star}}{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$$

TSA3
$$\frac{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{C}}{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{\star} \hookrightarrow e'}$$

C Complete translation for Thorn

C.1 Thorn synthetic translation for expressions

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t}}$ e translates to e' producing type t, with context Γ and class table K

THS1
$$\frac{\Gamma(x) = t}{\underline{K\ \Gamma} \vdash \underline{x} \hookrightarrow x \Uparrow \underline{t}}$$

THS2
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \qquad \overline{f(\overline{t_1}) : t_2 \in \mathrm{mtypes}(C, K) \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4}}{\underline{K\ \Gamma} \vdash \underline{e_1.f(\overline{e_2})} \hookrightarrow e_3.f(\overline{e_4}) \Uparrow \underline{t_2}}$$

THS3
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{?C} \qquad \underline{m(D_1) : D_2 \in \mathrm{mtypes}(C, K)} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{D_1} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow^{<D>} e_3@m(<\star>\ e_4) \Uparrow \underline{D_2}}$$

THS4
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{?C} \qquad \underline{m(t_1) : t \in \mathrm{mtypes}(C, K)} \qquad t \neq D \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m(e_4) \Uparrow \underline{t}}$$

THS5
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{D} \qquad \underline{m(t) : t' \in \mathrm{mtypes}(D, K)} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t} \hookrightarrow e_3}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3.m_{t \to t'}(e_4) \Uparrow \underline{t'}}$$

THS6
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{\star} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{\star} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m(e_4) \Uparrow \underline{\star}}$$

THS7
$$\frac{\overline{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{t} \hookrightarrow e_2} \qquad \textbf{class}\ C\,\{\,\overline{f : t}\ \overline{md}\,\}}{\underline{K\ \Gamma} \vdash \underline{\textbf{new}\ C(\overline{e_1})} \hookrightarrow \textbf{new}\ C(\overline{e_2}) \Uparrow \underline{C}}$$

C.2 Thorn analytic translation for expressions

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$ e translates to e' against type t, with context Γ and class table K

$$\textsc{tha1}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_2 \Uparrow \underline{C_2} \qquad \underline{K \cdot \vdash C_2 <:_t C_1}}{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{C_1} \hookrightarrow e_2}$$

$$\textsc{tha2}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_2 \Uparrow \underline{?D} \qquad \underline{K \cdot \vdash D <:_t C}}{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{C} \hookrightarrow <C>\, e_2}$$

$$\textsc{tha3}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_2 \Uparrow \underline{\star}}{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{C} \hookrightarrow <C>\, e_2}$$

$$\textsc{tha4}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_2 \Uparrow \underline{\star}}{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{?C} \hookrightarrow e_2}$$

$$\textsc{tha5}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_2 \Uparrow \underline{t} \qquad t \neq \star}{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{\star} \hookrightarrow e_2}$$

D Complete translation for Transient

D.1 Transient translation for methods

$\boxed{\underline{K\ C} \vdash \underline{md} \hookrightarrow_m md\ K}$ Transient translation for methods

$$\textsc{mtu}\quad \dfrac{\underline{K\ this\colon C\ x\colon \star} \vdash \underline{e} \Downarrow \underline{\star} \hookrightarrow e'}{\underline{K\ C} \vdash \underline{m(x\colon \star)\colon \star\ \{e\}} \hookrightarrow_m m(x\colon \star)\colon \star\ \{e'\}}$$

$$\textsc{mtt}\quad \dfrac{\underline{K\ this\colon C\ x\colon C} \vdash \underline{e} \Downarrow \underline{C'} \hookrightarrow e'}{\underline{K\ C} \vdash \underline{m(x\colon C)\colon C'\ \{e\}} \hookrightarrow_m m(x\colon \star)\colon \star\ \{<\star>\ \mathbf{new}\ A2(<C>\, x, e).f2()\}}$$

D.2 Transient synthetic translation for expressions

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t}}$ Transient translation for expressions

$$\textsc{grs1}\quad \dfrac{E(x) = t}{\underline{K\ E} \vdash \underline{x} \hookrightarrow <t>\, x \Uparrow \underline{t}}$$

$$\textsc{grs2}\quad \dfrac{\begin{array}{c}\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \\ m(t_1) : t_2 \in mtypes(C, K) \qquad \overline{\underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4}\end{array}}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow \prec t_2 \succ e_3.m_{\star\to\star}(<\star>\, e_4) \Uparrow \underline{t_2}}$$

$$\textsc{grs3}\quad \dfrac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{\star} \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{\star} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m_u(e_4) \Uparrow \underline{\star}}$$

$$\textsc{grs4}\quad \dfrac{\begin{array}{c}E(this) = C \qquad f(\overline{t_1}) : t_2 \in mtypes(C, K) \\ \underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t_1} \hookrightarrow e'\end{array}}{\underline{K\ \Gamma} \vdash \underline{this.f(\overline{e})} \hookrightarrow this.f(e') \Uparrow \underline{\star}}$$

$$\textsc{grs5}\quad \dfrac{\overline{\underline{K\ E} \vdash \underline{e_1} \Downarrow \underline{t} \hookrightarrow e_2} \qquad \mathbf{class}\ C\,\{\,\overline{f\colon t}\ \overline{md}\,\}}{\underline{K\ \Gamma} \vdash \underline{\mathbf{new}\ C(\overline{e_1})} \hookrightarrow \mathbf{new}\ C(\overline{e_2}) \Uparrow \underline{C}}$$

D.3 Transient analytic translation for expressions

$\boxed{\underline{K\ E} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$ Transient translation for bidirectional expressions

GRA1
$$\frac{\underline{K\ E} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash t' <: t}{\underline{K\ E} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$$

GRA2
$$\frac{\underline{K\ E} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t'} \qquad K \vdash t' \sim t}{\underline{K\ E} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow <t>\ e'}$$

E Complete translation for Typed Racket

E.1 Type Racket translation for program, class, and method

$\boxed{\underline{e\ K} \hookrightarrow_p e'\ K'}$ Type Racket translation for programs

$\boxed{\underline{K} \vdash \underline{\mathbf{class}\ C\,\{\ldots\ \ldots\}\ K} \hookrightarrow_c \mathbf{class}\ C\,\{\ldots\ \ldots\}\ K}$ Type Racket translation for classes

$\boxed{\underline{K\ C} \vdash \underline{md} \hookrightarrow_m md'\ K}$ Type Racket translation for methods

PT
$$\frac{\underline{K} \vdash \underline{K} \hookrightarrow_c K' \qquad \underline{K\ \cdot} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t}}{\underline{e\ K} \hookrightarrow_p e'\ K'}$$

CR1
$$\frac{\overline{\underline{K\ C} \vdash \underline{md} \hookrightarrow_m md'} \qquad \underline{K} \vdash \underline{K'} \hookrightarrow_c K''}{\underline{K} \vdash \underline{\mathbf{class}\ C\,\{\,\overline{f: t}\ \overline{md}\,\}\ K'} \hookrightarrow_c \mathbf{class}\ C\,\{\,\overline{f: t}\ \overline{md'}\,\}\ K''}$$

CR2
$$\frac{}{\underline{K} \vdash \underline{\cdot} \hookrightarrow_c \cdot}$$

MTT
$$\frac{\underline{K\ this: C\ x: t} \vdash \underline{e} \Downarrow \underline{t'} \hookrightarrow e'}{\underline{K\ C} \vdash \underline{m(x: t): t'\ \{e\}} \hookrightarrow_m m(x: t): t'\ \{e'\}}$$

E.2 Typed Racket synthetic translation

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \hookrightarrow e' \Uparrow \underline{t}}$ Typed Racket synthetic translation

TRS1
$$\frac{\Gamma(x) = t}{\underline{K\ \Gamma} \vdash \underline{x} \hookrightarrow x \Uparrow \underline{t}}$$

TRS2
$$\frac{\begin{array}{c}\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \\ f(\overline{t_1}) : t_2 \in mtypes(C, K) \qquad \overline{\underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t_1} \hookrightarrow e_4}\end{array}}{\underline{K\ \Gamma} \vdash \underline{e_1.f(\overline{e_2})} \hookrightarrow e_3.f(\overline{e_4}) \Uparrow \underline{t_2}}$$

TRS3
$$\frac{\begin{array}{c}\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \underline{C} \\ m(t) : t' \in mtypes(C, K) \qquad \overline{\underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{t} \hookrightarrow e_4}\end{array}}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3.m_{t \to t'}(e_4) \Uparrow \underline{C''}}$$

TRS5
$$\frac{\underline{K\ \Gamma} \vdash \underline{e_1} \hookrightarrow e_3 \Uparrow \star \qquad \underline{K\ \Gamma} \vdash \underline{e_2} \Downarrow \underline{\star} \hookrightarrow e_4}{\underline{K\ \Gamma} \vdash \underline{e_1.m(e_2)} \hookrightarrow e_3@m(e_4) \Uparrow \underline{\star}}$$

TRS6
$$\frac{\overline{\underline{K\ \Gamma} \vdash \underline{e_1} \Downarrow \underline{t} \hookrightarrow e_2} \qquad \mathbf{class}\ C\,\{\,\overline{f: t}\ \overline{md}\,\}}{\underline{K\ \Gamma} \vdash \underline{\mathbf{new}\ C(\overline{e_1})} \hookrightarrow \mathbf{new}\ C(\overline{e_2}) \Uparrow \underline{C}}$$

E.3 Typed Racket analytic translation

$\boxed{\underline{K\ \Gamma} \vdash \underline{e} \Downarrow \underline{t} \hookrightarrow e'}$ Typed Racket analytic translation

$$\text{TRA1}\quad \frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}'} \qquad \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \mathsf{e}'} \qquad\qquad \text{TRA2}\quad \frac{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \hookrightarrow \mathsf{e}' \Uparrow \underline{\mathsf{t}'} \qquad \mathsf{t} \neq \mathsf{t}'}{\underline{\mathsf{K}\ \Gamma} \vdash \underline{\mathsf{e}} \Downarrow \underline{\mathsf{t}} \hookrightarrow \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}'}$$

F Generative Behavioural Casts

F.1 Behavioural cast static and dynamic rules

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

$$\mathsf{K} \quad \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{a}\ \sigma \!\rightarrow \mathsf{K}'\ \mathsf{a}'\ \sigma' \qquad \mathsf{behcast}(\mathsf{a}, \mathsf{t}, \sigma, \mathsf{K}) = \mathsf{K}'\ \mathsf{a}'\ \sigma'$$

$$\mathcal{E} ::= \ldots \mid \ \blacktriangleleft \mathsf{t} \blacktriangleright \mathcal{E}$$

$$\text{BC1}\quad \frac{\begin{array}{c}\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}_1}\} \qquad \mathsf{D}\ \text{fresh} \qquad \mathsf{a}'\ \text{fresh} \qquad \mathsf{names}(\mathsf{mtypes}(\mathsf{C}', \mathsf{K})) \subseteq \mathsf{names}(\mathsf{mtypes}(\mathsf{C}, \mathsf{K})) \\ \mathsf{k} = \mathsf{wrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}', \mathsf{K}), \mathsf{D}) \qquad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}]\end{array}}{\mathsf{behcast}(\mathsf{a}, \mathsf{C}', \sigma, \mathsf{K}) = \mathsf{K}\ \mathsf{k}\ \mathsf{a}'\ \sigma'}$$

$$\text{BC2}\quad \frac{\begin{array}{c}\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}_1}\} \\ \mathsf{D}\ \text{fresh} \qquad \mathsf{a}'\ \text{fresh} \qquad \mathsf{k} = \mathsf{wrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C}, \mathsf{K}), \mathsf{mtypes}(\mathsf{C}, \mathsf{K}), \mathsf{D}) \qquad \sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}]\end{array}}{\mathsf{behcast}(\mathsf{a}, \star, \sigma, \mathsf{K}) = \mathsf{K}\ \mathsf{k}\ \mathsf{a}'\ \sigma'}$$

Figure 22 Behavioural casts.

F.2 Class wrapper for Behavioural semantics

The wrap function is used by the Type Racket language to generate the wrapper classes when encountering typed code.

$$\boxed{\mathsf{mt} \ ::= \mathsf{m}(\mathsf{t}) : \mathsf{t} \ \mid \ \mathsf{f}(\mathsf{t}) : \mathsf{t} \ \mid \ \mathsf{f}() : \mathsf{t}} \quad \text{method typing}$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{wrap}(\mathsf{C}, \overline{\mathsf{md}}, \overline{\mathsf{mt}}, \overline{\mathsf{mt}'}, \mathsf{D}) =} \\
\quad \textbf{class}\ \mathsf{D}\ \{ & \\
\quad\quad \mathsf{that}\colon \mathsf{C} & \\
\quad\quad \mathsf{f}()\colon \mathsf{t}'\ \{\ \blacktriangleleft \mathsf{t}' \blacktriangleright \mathsf{this.that}().\mathsf{f}()\ \} & \mathsf{f}()\colon \mathsf{t} \in \overline{\mathsf{mt}}\ \wedge\ \mathsf{f}()\colon \mathsf{t}' \in \overline{\mathsf{mt}'} \\
\quad\quad \mathsf{f}(\mathsf{x}\colon \mathsf{t}')\colon \mathsf{t}'\ \{\ \blacktriangleleft \mathsf{t}' \blacktriangleright \mathsf{this.that}().\mathsf{f}(\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x})\ \} & \mathsf{f}(\mathsf{t})\colon \mathsf{t} \in \overline{\mathsf{mt}}\ \wedge\ \mathsf{f}(\mathsf{t}')\colon \mathsf{t}' \in \overline{\mathsf{mt}'} \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\ \blacktriangleleft \mathsf{t}' \blacktriangleright \mathsf{e}\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \mathsf{C}')\colon \mathsf{C}''\ \{\mathsf{e}\} \in \overline{\mathsf{md}}\ \wedge\ \mathsf{m}(\mathsf{t})\colon \mathsf{t}' \in \overline{\mathsf{mt}'} \\
 & \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{this.m}(\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x})\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \mathsf{C}')\colon \mathsf{C}''\ \{\mathsf{e}\} \in \overline{\mathsf{md}}\ \wedge\ \mathsf{m}(\mathsf{t})\colon \mathsf{t}' \in \overline{\mathsf{mt}'} \\
 & \ \wedge\ \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{e}'\} \notin \overline{\mathsf{md}}\ \wedge\ \mathsf{t} \neq \star \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}'\ \{\ \blacktriangleleft \mathsf{t}' \blacktriangleright \mathsf{e}\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{e}\} \in \overline{\mathsf{md}}\ \wedge\ \mathsf{m}(\mathsf{t})\colon \mathsf{t}' \in \overline{\mathsf{mt}'} \\
 & \\
\quad\quad \mathsf{f}()\colon \mathsf{t}\ \{\ \mathsf{this.that}().\mathsf{f}()\ \} & \mathsf{f}()\colon \mathsf{t} \in \overline{\mathsf{mt}}\ \wedge\ \mathsf{f}()\colon \mathsf{t}' \notin \overline{\mathsf{mt}'} \\
\quad\quad \mathsf{f}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\ \mathsf{this.that}().\mathsf{f}(\mathsf{x})\ \} & \mathsf{f}(\mathsf{t})\colon \mathsf{t} \in \overline{\mathsf{mt}}\ \wedge\ \mathsf{f}(\mathsf{t}')\colon \mathsf{t}' \notin \overline{\mathsf{mt}'} \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \mathsf{C}')\colon \mathsf{C}''\ \{\ \mathsf{e}\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \mathsf{C}')\colon \mathsf{C}''\ \{\mathsf{e}\} \in \overline{\mathsf{md}}\ \wedge\ \mathsf{m}(\mathsf{t})\colon \mathsf{t}' \notin \overline{\mathsf{mt}'} \\
 & \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \mathsf{e}\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\mathsf{e}\} \in \overline{\mathsf{md}}\ \wedge\ \mathsf{m}(\mathsf{t})\colon \mathsf{t}' \notin \overline{\mathsf{mt}'} \\
 & \\
\quad \} & \\
\multicolumn{2}{l}{\mathsf{wrap}(\mathsf{C}, \overline{\mathsf{md}}, \overline{\mathsf{mt}}, \mathsf{D}) =} \\
\quad \textbf{class}\ \mathsf{D}\ \{ & \\
\quad\quad \mathsf{that}\colon \mathsf{C} & \\
\quad\quad \mathsf{f}()\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{this.that}().\mathsf{f}()\ \} & \mathsf{f}()\colon \mathsf{t} \in \overline{\mathsf{mt}} \\
\quad\quad \mathsf{f}(\mathsf{x}\colon \star)\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{this.that}().\mathsf{f}(\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{x})\ \} & \mathsf{f}(\mathsf{t})\colon \mathsf{t} \in \overline{\mathsf{mt}} \\
\quad\quad \mathsf{m}(\mathsf{x}\colon \star)\colon \star\ \{\ \blacktriangleleft \star \blacktriangleright \mathsf{e}\ \} & \forall\ \mathsf{m}\ .\ \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\} \in \overline{\mathsf{md}} \\
 & \\
\quad \} &
\end{array}
$$

F.3 Static function

The static function returns true if the class D does not (transitively) contain any $\star$ type in any of its fields or methods.

$\boxed{\mathsf{static}(\mathsf{D}, \mathsf{K}, \overline{\mathsf{C}}) = \texttt{Bool}}$ static function

$$
\textsc{st1}\ \frac{}{\mathsf{static}(\star, \mathsf{K}, \overline{\mathsf{C}}) = \texttt{False}}
\qquad
\textsc{st2}\ \frac{\mathsf{D} \in \overline{\mathsf{C}}}{\mathsf{static}(\mathsf{D}, \mathsf{K}, \overline{\mathsf{C}}) = \texttt{True}}
\qquad
\textsc{st3}\ \frac{\mathsf{D}\ \text{empty}}{\mathsf{static}(\mathsf{D}, \mathsf{K}, \overline{\mathsf{C}}) = \texttt{True}}
$$

$$
\textsc{st4}\ \frac{\textbf{class}\ \mathsf{C}\ \{\ \overline{\mathsf{f}\colon \mathsf{t}}\ \overline{\mathsf{md}}\ \} \in \mathsf{K} \qquad \mathsf{signature}(\overline{\mathsf{md}}) = \overline{\mathsf{m}(\mathsf{t}')\colon \mathsf{t}''} \qquad \overline{\mathsf{C}'} = \overline{\mathsf{C}}, \mathsf{D}}{\mathsf{static}(\mathsf{D}, \mathsf{K}, \overline{\mathsf{C}}) = \mathsf{static}(\overline{\mathsf{t}}, \mathsf{K}, \overline{\mathsf{C}'}) \cap \mathsf{static}(\overline{\mathsf{t}'}, \mathsf{K}, \overline{\mathsf{C}'}) \cap \mathsf{static}(\overline{\mathsf{t}''}, \mathsf{K}, \overline{\mathsf{C}'})}
$$

F.4 Wftype function

The function $\mathsf{wftype}(\overline{\mathsf{f}}, \mathsf{C}, \mathsf{K})$ denotes the function that looks up the type of a particular set of fields in C.

$$\mathsf{wftype}(\bar{\mathsf{f}}, \mathsf{C}, \mathsf{K}) = \bar{\mathsf{t}} \;\; \textit{if} \;\; \begin{cases} \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f}' : \mathsf{t}'}\ \overline{\mathsf{md}}\,\} \in \mathsf{K} \\ \bar{\mathsf{t}} = \{\bar{\mathsf{t}} \subseteq \overline{\mathsf{t}'} \mid \forall\ \mathsf{f} \in \bar{\mathsf{f}}\ .\ \mathsf{f} \in \mathsf{names}(\overline{\mathsf{f}' : \mathsf{t}'})\} \end{cases}$$

F.5 Mtype function

The `mtypes` function takes a class name C and the class table K, and outputs a list of typing signatures $\overline{\mathsf{mt}}$ for every method in class C, which includes the implicit getter and setter methods for every field in the definition of class C. (**Note**: An user cannot define a getter or setter method for any field that already exists in the class. Similarly, a field cannot be declared in a class that already has a getter or setter method for that field. This is enforced by the `overloading` function in class well-formedness.)

$$\mathsf{mtypes}(\mathsf{C}, \mathsf{K}) = \overline{\mathsf{mt}} \;\; \textit{if} \;\; \begin{cases} \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K} \\ \overline{\mathsf{mt}} = \mathsf{signature}(\overline{\mathsf{md}}) \oplus \forall\ \mathsf{f} : \mathsf{t} \in \overline{\mathsf{f} : \mathsf{t}} \mid \mathsf{f} \notin \mathsf{names}(\overline{\mathsf{md}})\ \wedge\ \mathsf{f} \neq \mathsf{that}\ .\ \mathsf{typing}(\mathsf{f} : \mathsf{t}) \end{cases}$$

F.6 getmds function

The function getmds(C, K) denotes the function that returns the method definitions inside the class C.

$$\mathsf{getmds}(\mathsf{C}, \mathsf{K}) = \overline{\mathsf{md}} \;\; \textit{if} \;\; \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K}$$

F.7 Ftype function

The function ftypes(a,C,σ,K)returns the old references and the new types for them according to the new wrapper C.

$$\frac{\sigma(\mathsf{a}) = \mathsf{D}\{\mathsf{a}'\} \qquad \sigma(\mathsf{a}') = \mathsf{E}\{\overline{\mathsf{a}''}\} \qquad \textbf{class}\ \mathsf{E}\,\{\,\overline{\mathsf{f} : \mathsf{t}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K} \qquad \mathsf{wftype}(\bar{\mathsf{f}}, \mathsf{C}, \mathsf{K}) = \overline{\mathsf{t}'}}{\mathsf{ftypes}(\mathsf{a}, \mathsf{C}, \sigma, \mathsf{K}) = \overline{\mathsf{a}''}\ \overline{\mathsf{t}'}}\ \text{FT1}$$

F.8 Dynamic function

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

$$\frac{\mathsf{dyn}(\overline{\mathsf{mt}}) = \overline{\mathsf{mt}'}}{\mathsf{dyn}(\mathsf{m}(\mathsf{t}) : \mathsf{t}\ \ \overline{\mathsf{mt}}) = \mathsf{m}(\star) : \star\ \ \overline{\mathsf{mt}'}}\ \text{DYN1} \qquad \frac{\mathsf{dyn}(\overline{\mathsf{mt}}) = \overline{\mathsf{mt}'}}{\mathsf{dyn}(\mathsf{f}(\mathsf{t}) : \mathsf{t}\ \ \overline{\mathsf{mt}}) = \mathsf{f}(\star) : \star\ \ \overline{\mathsf{mt}'}}\ \text{DYN2}$$

$$\frac{\mathsf{dyn}(\overline{\mathsf{mt}}) = \overline{\mathsf{mt}'}}{\mathsf{dyn}(\mathsf{f}() : \mathsf{t}\ \ \overline{\mathsf{mt}}) = \mathsf{f}() : \star\ \ \overline{\mathsf{mt}'}}\ \text{DYN3} \qquad \frac{}{\mathsf{dyn}(\cdot) = \cdot}\ \text{DYNE}$$

F.9 Signature function

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

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

SG1
$$\frac{\mathsf{md} = \mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\} \qquad \mathsf{signature}(\overline{\mathsf{md}}) = \overline{\mathsf{mt}}}{\mathsf{signature}(\mathsf{md}\,\overline{\mathsf{md}}) = \mathsf{m}(\mathsf{t})\colon \mathsf{t}\ \ \overline{\mathsf{mt}}}$$

SG2
$$\frac{\mathsf{md} = \mathsf{f}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\} \qquad \mathsf{signature}(\overline{\mathsf{md}}) = \overline{\mathsf{mt}}}{\mathsf{signature}(\mathsf{md}\,\overline{\mathsf{md}}) = \mathsf{f}(\mathsf{t})\colon \mathsf{t}\ \ \overline{\mathsf{mt}}}$$

SG3
$$\frac{\mathsf{md} = \mathsf{f}()\colon \mathsf{t}\ \{\mathsf{e}\} \qquad \mathsf{signature}(\overline{\mathsf{md}}) = \overline{\mathsf{mt}}}{\mathsf{signature}(\mathsf{md}\,\overline{\mathsf{md}}) = \mathsf{f}()\colon \mathsf{t}\ \ \overline{\mathsf{mt}}}$$

F.10 Typing function

The typing function takes field definitions and returns the method typing signature of that field's getter and setter methods.

TY1
$$\frac{}{\mathsf{typing}(\mathsf{f}\colon \mathsf{t}) = \mathsf{f}(\mathsf{t})\colon \mathsf{t}\ \ \mathsf{f}()\colon \mathsf{t}}$$

F.11 Names function

The Names function takes either a field definition, method definition, or method typing, and returns the name of the respective field/method.

NE
$$\frac{}{\mathsf{names}(\emptyset) = \emptyset}$$

$\boxed{\mathsf{names}(\overline{\mathsf{fd}})}$ field naming

NF
$$\frac{\overline{\mathsf{x}} = \mathsf{f}, \mathsf{names}(\overline{\mathsf{fd}})}{\mathsf{names}(\mathsf{f}\colon \mathsf{t}, \overline{\mathsf{fd}}) = \overline{\mathsf{x}}}$$

$\boxed{\mathsf{names}(\overline{\mathsf{md}})}$ method naming

NM1
$$\frac{\overline{\mathsf{x}} = \mathsf{m}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{m}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\}, \overline{\mathsf{md}}) = \overline{\mathsf{x}}}$$

NM2
$$\frac{\overline{\mathsf{x}} = \mathsf{f}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{f}(\mathsf{x}\colon \mathsf{t})\colon \mathsf{t}\ \{\mathsf{e}\}, \overline{\mathsf{md}}) = \overline{\mathsf{x}}}$$

NM3
$$\frac{\overline{\mathsf{x}} = \mathsf{f}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{f}()\colon \mathsf{t}\ \{\mathsf{e}\}, \overline{\mathsf{md}}) = \overline{\mathsf{x}}}$$

$\boxed{\mathsf{names}(\overline{\mathsf{mt}})}$ type naming

NMT1
$$\frac{\overline{\mathsf{x}} = \mathsf{m}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{m}(\mathsf{t})\colon \mathsf{t}, \overline{\mathsf{mt}}) = \overline{\mathsf{x}}}$$

NMT2
$$\frac{\overline{\mathsf{x}} = \mathsf{f}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{f}(\mathsf{t})\colon \mathsf{t}, \overline{\mathsf{mt}}) = \overline{\mathsf{x}}}$$

NMT3
$$\frac{\overline{\mathsf{x}} = \mathsf{f}, \mathsf{names}(\overline{\mathsf{md}})}{\mathsf{names}(\mathsf{f}()\colon \mathsf{t}, \overline{\mathsf{mt}}) = \overline{\mathsf{x}}}$$

G Source language syntax and semantics

G.1 Syntax

$$\begin{array}{rcl} e & ::= & x \mid \text{this} \\ & \mid & \textbf{new}\ C(\overline{e}) \mid e.f() \mid e.f(e) \\ & \mid & e.m(e) \mid a \end{array} \qquad \begin{array}{rcl} k & ::= & \textbf{class}\ C\,\{\,\overline{fd}\ \overline{md}\,\} \\ md & ::= & m(x\colon t)\colon t\ \{e\} \mid f(x\colon t)\colon t\ \{e\} \mid f()\colon t\ \{e\} \\ t & ::= & \star \mid C \mid ?C \\ fd & ::= & f\colon t \end{array}$$

Figure 23 Source language syntax.

G.2 Thorn subtyping

$$\textsc{SWeak}\quad \frac{\mu\ K \vdash_{th} C \mathrel{\underline{<:}} D}{\mu\ K \vdash_{th} ?C \mathrel{\underline{<:}} ?D} \qquad \textsc{SLow}\quad \frac{\mu\ K \vdash_{th} C \mathrel{\underline{<:}} D}{\mu\ K \vdash_{th} C \mathrel{\underline{<:}} ?D}$$

$$\textsc{SRef}\quad \frac{}{\mu\ K \vdash_{th} t \mathrel{\underline{<:}} t} \qquad \textsc{SAss}\quad \frac{C\underline{<:}D \in \mu}{\mu\ K \vdash_{th} C \mathrel{\underline{<:}} D}$$

$$\textsc{SRec}\quad \frac{\mu' = \mu\ C\underline{<:}D \qquad mt \in mtypes(D, K) \implies mt' \in mtypes(C, K)\ .\ \mu'\ K \vdash_{th} mt \mathrel{\underline{<:}} mt'}{\mu\ K \vdash_{th} C \mathrel{\underline{<:}} D}$$

$$\textsc{SMet}\quad \frac{\mu\ K \vdash_{th} t_1 \mathrel{\underline{<:}} t_2 \qquad \mu\ K \vdash_{th} t_2' \mathrel{\underline{<:}} t_1'}{\mu\ K \vdash_{th} m(t_1)\colon t_1' \mathrel{\underline{<:}} m(t_2)\colon t_2'} \qquad \textsc{SGet}\quad \frac{\mu\ K \vdash_{th} t_1 \mathrel{\underline{<:}} t_2}{\mu\ K \vdash_{th} f()\colon t_1 \mathrel{\underline{<:}} f()\colon t_2}$$

$$\textsc{SSet}\quad \frac{\mu\ K \vdash_{th} t_1 \mathrel{\underline{<:}} t_2}{\mu\ K \vdash_{th} f(t_1)\colon t_1 \mathrel{\underline{<:}} f(t_2)\colon t_2}$$

G.3 Thorn well-formedness

$\boxed{e\ K\ \checkmark_{th}}$ Well-formed program

$$\textsc{WP}\quad \frac{k \in K \implies \cdot\, K \vdash_{th}\ k\ \checkmark \qquad \Gamma\ \cdot\ K \vdash_{th} e : t}{e\ K\ \checkmark_{th}}$$

$\boxed{\sigma\, K \vdash_{th}\ \textbf{class}\ C\,\{\,\overline{fd}\ \overline{md}\,\}\ \checkmark}$ Well-formed class

$$\textsc{WC}\quad \frac{\begin{array}{c} overloading_{\emptyset}(\overline{fd}, \overline{md}) \\ fd \in \overline{fd} \implies K \vdash_{th}\ fd\ \checkmark \qquad md \in \overline{md} \implies this : C\ \ \sigma\, K \vdash_{th}\ md\ \checkmark \end{array}}{\sigma\, K \vdash_{th}\ \textbf{class}\ C\,\{\,\overline{fd}\ \overline{md}\,\}\ \checkmark}$$

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

$$\textsc{wt}\ \frac{\Gamma\ \mathsf{x}{:}\,\mathsf{C}\ \ \sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{D} \qquad \mathsf{K} \vdash_{th}\ \mathsf{C}\ \checkmark \qquad \mathsf{K} \vdash_{th}\ \mathsf{D}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th}\ \mathsf{m}(\mathsf{x}{:}\,\mathsf{C}){:}\,\mathsf{D}\ \{\mathsf{e}\}\ \checkmark}
\qquad
\textsc{wwt}\ \frac{\Gamma\ \mathsf{x}{:}\,?\mathsf{C}\ \ \sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : ?\mathsf{D} \qquad \mathsf{K} \vdash_{th}\ ?\mathsf{C}\ \checkmark \qquad \mathsf{K} \vdash_{th}\ ?\mathsf{D}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th}\ \mathsf{m}(\mathsf{x}{:}\,?\mathsf{C}){:}\,?\mathsf{D}\ \{\mathsf{e}\}\ \checkmark}$$

$$\textsc{wu}\ \frac{\Gamma\ \mathsf{x}{:}\,\star\ \ \sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \star}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th}\ \mathsf{m}(\mathsf{x}{:}\,\star){:}\,\star\ \{\mathsf{e}\}\ \checkmark}
\qquad
\textsc{ws}\ \frac{\Gamma\ \mathsf{x}{:}\,\mathsf{t}'\ \ \sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash_{th}\ \mathsf{t}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th}\ \mathsf{f}(\mathsf{x}{:}\,\mathsf{t}){:}\,\mathsf{t}\ \{\mathsf{e}\}\ \checkmark}$$

$$\textsc{wg}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{t} \qquad \mathsf{K} \vdash_{th}\ \mathsf{t}\ \checkmark}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th}\ \mathsf{f}(){:}\,\mathsf{t}\ \{\mathsf{e}\}\ \checkmark}$$

$\boxed{\mathsf{K} \vdash_{th}\ \mathsf{f}{:}\,\mathsf{t}\ \checkmark}$ Well-formed fields

$$\textsc{wf}\ \frac{\mathsf{K} \vdash_{th}\ \mathsf{t}\ \checkmark}{\mathsf{K} \vdash_{th}\ \mathsf{f}{:}\,\mathsf{t}\ \checkmark}$$

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

$$\textsc{wa}\ \frac{}{\mathsf{K} \vdash_{th}\ \star\ \checkmark}
\qquad
\textsc{wc}\ \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_{th}\ \mathsf{C}\ \checkmark}
\qquad
\textsc{ww}\ \frac{\mathsf{C} \in \mathsf{K}}{\mathsf{K} \vdash_{th}\ ?\mathsf{C}\ \checkmark}$$

G.4 Thorn expression typing

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

$$\textsc{tw1}\ \frac{\Gamma(\mathsf{x}) = \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{x} : \mathsf{t}}
\qquad
\textsc{tw2}\ \frac{}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{a} : \star}
\qquad
\textsc{tw3}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{t}' \qquad \mu\ \mathsf{K} \vdash_{th} \mathsf{t}' \underline{<:}\ \mathsf{t}}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{t}}$$

$$\textsc{tw4}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{C} \qquad \mathsf{f}(){:}\,\mathsf{t}' \in \mathsf{mtypes}(\mathsf{C},\mathsf{K})}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e.f}() : \mathsf{t}'}$$

$$\textsc{tw5}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{C} \qquad \mathsf{f}(\mathsf{t}'){:}\,\mathsf{t}' \in \mathsf{mtypes}(\mathsf{C},\mathsf{K}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e}' : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e.f}(\mathsf{e}') : \mathsf{t}'}$$

$$\textsc{tw6}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e} : \mathsf{C} \qquad \mathsf{m}(\mathsf{t}'){:}\,\mathsf{t}'' \in \mathsf{mtypes}(\mathsf{C},\mathsf{K}) \qquad \Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e}' : \mathsf{t}'}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e.m}(\mathsf{e}') : \mathsf{t}''}$$

$$\textsc{tw7}\ \frac{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e}_1 : \mathsf{t}_1 \ \ \ldots\ \ \Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{e}_n : \mathsf{t}_n \qquad \overline{\mathsf{fd}} = \mathsf{f}_1{:}\,\mathsf{t}_1\ \ldots\ \mathsf{f}_n{:}\,\mathsf{t}_n \qquad \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K}}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \textbf{new}\ \mathsf{C}(\mathsf{e}_1 \ldots \mathsf{e}_n) : \mathsf{C}}$$

$$\textsc{tw8}\ \frac{\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}'}\}}{\Gamma\,\sigma\,\mathsf{K} \vdash_{th} \mathsf{a} : \mathsf{C}}$$

H Proofs of Related Theorems

H.1 Accessory Lemmas

H.1.0.1 Evaluation Extends Class Tables

If $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$ then $\mathsf{K}' = \mathsf{K}\ \mathsf{K}''$ for some K''.

H.1.0.2 Weakening of Expression Typing

If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$ then $\Gamma\ \Gamma'\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$.
If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$ then $\Gamma\,\sigma\ \sigma'\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$.
If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$ then $\Gamma\,\sigma\,\mathsf{K}\ \mathsf{K}' \vdash \mathsf{e} : \mathsf{t}$.

H.1.0.3 Weakening of Well-formedness

If $\mathsf{K} \vdash \mathsf{t}\ \checkmark$ then $\mathsf{K}\ \mathsf{K}' \vdash \mathsf{t}\ \checkmark$.
If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{md}\ \checkmark$ then $\Gamma\ \Gamma'\,\sigma\,\mathsf{K} \vdash \mathsf{md}\ \checkmark$.
If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{md}\ \checkmark$ then $\Gamma\,\sigma\ \sigma'\,\mathsf{K} \vdash \mathsf{md}\ \checkmark$.
If $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{md}\ \checkmark$ then $\Gamma\,\sigma\,\mathsf{K}\ \mathsf{K}' \vdash \mathsf{md}\ \checkmark$.
If $\mathsf{K} \vdash \mathsf{f} : \mathsf{t}\ \checkmark$ then $\mathsf{K}\ \mathsf{K}' \vdash \mathsf{f} : \mathsf{t}\ \checkmark$.
If $\mathsf{K} \vdash \mathsf{t}\ \checkmark$ then $\mathsf{K}\ \mathsf{K}' \vdash \mathsf{t}\ \checkmark$.

H.1.0.4 Weakening of Subtyping

If $\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$ then $\mu\ \mathsf{K}\ \mathsf{K}' \vdash \mathsf{t} <: \mathsf{t}'$.
If $\mu\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$ then $\mu\ \mu'\ \mathsf{K} \vdash \mathsf{t} <: \mathsf{t}'$.

H.1.0.5 Weakening of mtypes

If $\mathsf{n}(\mathsf{x} : \mathsf{t}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})$, then $\mathsf{n}(\mathsf{x} : \mathsf{t}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}\ \mathsf{K}')$ for some K'.
If $\mathsf{f}() : \mathsf{t} \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})$, then $\mathsf{f}() : \mathsf{t} \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K}\ \mathsf{K}')$, for some K'.

H.1.0.6 Substitution

If $\overline{\mathsf{x} : \mathsf{t}'}\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$ and $\overline{\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{t}'}$, then $\cdot\,\sigma\,\mathsf{K} \vdash [\mathsf{a}/\mathsf{x}]\mathsf{e} : \mathsf{t}$

H.1.0.7 Correctness of mtypes(C, K)

If $\mathsf{m}(\mathsf{t}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})$, $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a}' : \mathsf{t}$, then $\mathsf{K}\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{a}')\ \sigma \to \mathsf{K}\ \mathsf{e}''\ \sigma$ where $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$.
If $\mathsf{f}(\overline{\mathsf{t}}) : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})$, $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a}' : \mathsf{t}$, then $\mathsf{K}\ \mathsf{a}.\mathsf{f}(\overline{\mathsf{a}'})\ \sigma \to \mathsf{K}\ \mathsf{e}''\ \sigma$ where $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$.
If $\mathsf{f}() : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C}, \mathsf{K})$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}$, then $\mathsf{K}\ \mathsf{a}.\mathsf{f}()\ \sigma \to \mathsf{K}\ \mathsf{e}''\ \sigma$ where $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$.

H.1.0.8 Canonical forms

If $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}$, then $\sigma[\mathsf{a} \mapsto \mathsf{C}\{\overline{\mathsf{a}}\}]$.

H.1.0.9 Evaluation retains typing

If $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$, $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t}'$, and $\mathsf{K}\ \mathsf{e}'\ \sigma\ \checkmark$, then if $\mathsf{K}\ \mathsf{e}'\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}''\ \sigma'$, it follows that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e} : \mathsf{t}$ and $\mathsf{K}' = \mathsf{K}\ \mathsf{K}''$.

H.2 Reduction preserves Well-formedness

If $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$ and $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$, then $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$.

H.3 Consistent Class Table

If $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$, then $\forall\ \textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\}\ .\ \mathsf{K} \vdash \overline{\mathsf{fd}}\ \checkmark\ \wedge\ \cdot\,\sigma\,\mathsf{K} \vdash \overline{\mathsf{md}}\ \checkmark$

H.4 Correctness of wrap

If $\mathsf{K}\ \ \checkmark$ (TODO), if $\mathsf{C} \in \mathsf{K}$, $\mathsf{C}' \in \mathsf{K}$, D free, $\overline{\mathsf{md}} = \mathsf{getmds}(\mathsf{C},\mathsf{K})$, $\overline{\mathsf{mt}} = \mathsf{mtypes}(\mathsf{C},\mathsf{K})$, $\overline{\mathsf{mt}'} = \mathsf{mtypes}(\mathsf{C}',\mathsf{K})$, and $\mathsf{k} = \mathsf{wrap}(\mathsf{C}, \overline{\mathsf{md}}, \overline{\mathsf{mt}}, \overline{\mathsf{mt}'}, \mathsf{D})$, then $\cdot\ \mathsf{K}\ \mathsf{k} \vdash \mathsf{D} <: \mathsf{C}'$ and $\mathsf{k}\ \ \checkmark$.

H.5 Type Soundness of Core KafKa Typing

Given that $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$, then either there is some e' such that $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$ and $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$ and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$ hold, or e is stuck in one of the following forms:

- a
- $\mathcal{E}[\mathsf{a}@\mathsf{m}(\mathsf{a}')]$
- $\mathcal{E}[<\mathsf{t}'>\mathsf{a}]$
- $\mathcal{E}[\prec \mathsf{t}' \succ \mathsf{a}]$

We proceed with rule induction on the judgement used to conclude $\Gamma\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$. Note that we refer to rule preconditions from left to right.

- W1 Not applicable, since $\Gamma = \cdot$ and therefore contains no variables.
- W2
  We apply the IH to the first precondition. If we get stuck in the IH, then the entire expression gets stuck or terminates, trivially. Therefore, the interesting case is when $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$ and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}'$. Since, by the second precondition, we know that $\mu\ \mathsf{K} \vdash \mathsf{t}' <: \mathsf{t}$, it follows by weakening of subtyping $\mu\ \mathsf{K}' \vdash \mathsf{t}' <: \mathsf{t}$. We can then apply W2 to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$, and therefore $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$, and the theorem holds.
- W3
  We apply the IH to the first precondition, finding that either e is a value a, e is stuck, or $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{C}$.
  If e steps to some e', and since we know that $\mathsf{f}() : \mathsf{t}' \in \mathsf{mtypes}(\mathsf{C},\mathsf{K})$, we can then apply W3 via weakening of mtypes.
  If e is a value a, then we apply correctness of mtypes to find that $\mathsf{K}\ \mathsf{a}.\mathsf{f}()\ \sigma \rightarrow \mathsf{K}\ \mathsf{e}'\ \sigma$ for some e', and that $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \mathsf{t}$. Therefore, the theorem holds.
  If e is a stuck state of the form $\mathcal{E}[\mathsf{e}']$, define $\mathcal{E}' = \mathcal{E}[\mathsf{e}'].\mathsf{f}()$, and then the theorem holds.
- W4
  We apply the IH to the first precondition, finding that either e is a value a, e is a stuck state, or $\mathsf{K}\ \mathsf{e}\ \sigma \rightarrow \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{C}$, then case analyze.

  - $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{C}$. Apply H.1.0.9 to the second precondition to find that $\mathsf{K}'\,\sigma'\,\mathsf{e}' \vdash \mathsf{t}$:, and apply W4 to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}''.\mathsf{f}(\mathsf{e}') : \mathsf{t}'$.
  - e is a: Apply the IH to e'. Then, either e' is a value a', e' is stuck, or $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{t}$.
    * $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{t}$. Then, $\mathsf{K}\ \mathsf{a}.\mathsf{f}(\mathsf{e}')\ \sigma \to \mathsf{K}'\ \mathsf{a}.\mathsf{f}(\mathsf{e}'')\ \sigma'$ by the definition of evaluation contexts. Apply H.1.0.9 to the first precondition to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a} : \mathsf{C}$, then use W4 to conclude that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a}.\mathsf{f}(\mathsf{e}'') : \mathsf{t}'$.
    * $\mathsf{e}' = \mathsf{a}'$. In this case, apply correctness of mtypes to find that $\mathsf{K}\ \mathsf{a}.\mathsf{f}(\mathsf{a}')\ \sigma \to \mathsf{K}\ \mathsf{e}''\ \sigma$, where $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$.
    * e' is stuck at one of the three stuck states, of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathsf{a}.\mathsf{f}(\mathcal{E}[\mathsf{e}''])$, which is stuck.
  - If e is stuck at one of the three stuck states, of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathcal{E}[\mathsf{e}''].\mathsf{f}(\mathsf{e}')$, which is stuck.

- W5
  We apply the IH to the first precondition, finding that either e is a value a, e is a stuck state, or $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{C}$, then case analyze.
  - $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{C}$. Apply H.1.0.9 to the second precondition to find that $\mathsf{K}'\,\sigma'\,\mathsf{e}' \vdash \mathsf{t}$:, and apply W5 to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}''.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}') : \mathsf{t}'$.
  - e is a: Apply the IH to e'. Then, either e' is a value a', e' is stuck, or $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{t}$.
    * $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \mathsf{C}$. Then, $\mathsf{K}\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')\ \sigma \to \mathsf{K}'\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}'')\ \sigma'$ by the definition of evaluation contexts. Apply H.1.0.9 to the first precondition to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a} : \mathsf{C}$, then use W5 to conclude that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}'') : \mathsf{t}'$.
    * $\mathsf{e}' = \mathsf{a}'$. In this case, apply correctness of mtypes to find that $\mathsf{K}\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{a}')\ \sigma \to \mathsf{K}\ \mathsf{e}''\ \sigma$, where $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}'' : \mathsf{t}'$.
    * e' is stuck at one of the three stuck states, of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathcal{E}[\mathsf{e}''])$, which is stuck.
  - If e is stuck at one of the three stuck states of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathcal{E}[\mathsf{e}''].\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')$, which is stuck.

- W6
  We apply the IH to the first precondition, finding that either e is a value a, e is a stuck state, or $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \star$, then case analyze.
  - $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \star$. Apply H.1.0.9 to the second precondition to find that $\mathsf{K}'\,\sigma'\,\mathsf{e}' \vdash \star$:, and apply W6 to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}''@\mathsf{m}(\mathsf{e}') : \star$.
  - e is a: Apply the IH to e'. Then, either e' is a value a', e' is stuck, or $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \star$.
    * $\mathsf{K}\ \mathsf{e}'\ \sigma \to \mathsf{K}'\ \mathsf{e}''\ \sigma'$, $\mathsf{K}'\ \mathsf{e}''\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'' : \star$. Then, $\mathsf{K}\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}')\ \sigma \to \mathsf{K}'\ \mathsf{a}.\mathsf{m}_{\mathsf{t}\to\mathsf{t}'}(\mathsf{e}'')\ \sigma'$ by the definition of evaluation contexts. Apply H.1.0.9 to the first precondition to find that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a} : \star$, then use W6 to conclude that $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a}@\mathsf{m}(\mathsf{e}'') : \star$.
    * $\mathsf{e}' = \mathsf{a}'$. If $\sigma(a) = \mathsf{C}\{\overline{\mathsf{a}''}\}$, and if $\mathsf{m}(\mathsf{x} : \star) : \star\ \{\mathsf{e}\} \in \mathsf{K}(\mathsf{C})$, then $\mathsf{K}\ \mathsf{a}@\mathsf{m}(\mathsf{a}')\ \sigma \to \mathsf{K}\ [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e}\ \star$, by the definition of evaluation. Then, since $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$, it follows that $\mathsf{C}\ \checkmark$, and then that $\mathsf{this} : \mathsf{C}\ \mathsf{x} : \star\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \star$. Then, since we have that $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a} : \mathsf{C}$ (by application of W10), and that $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}' : \star$, we can use the substitution lemma to find that $\cdot\,\sigma\,\mathsf{K} \vdash [\mathsf{a}/\mathsf{this}\ \mathsf{a}'/\mathsf{x}]\mathsf{e} : \star$, and the theorem holds.

If the precondition does not apply, and C does not contain m under type $\star$, then the expression gets stuck, and the theorem holds.

- e' is stuck at one of the three stuck states, of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathsf{a@m}(\mathcal{E}[\mathsf{e}''])$, which is stuck.

- If e is stuck at one of the three stuck states of the form $\mathcal{E}[\mathsf{e}'']$, then we can construct $\mathcal{E}'[\mathsf{e}''] = \mathcal{E}[\mathsf{e}'']\mathsf{@m}(\mathsf{e}')$, which is stuck.

- W7
  We apply the IH to the preconditions $\mathsf{e}_1 \cdots \mathsf{e}_n$, finding that either e_i is a value a, e_i is a stuck state, or $\mathsf{K}\ \mathsf{e}_i\ \sigma \to \mathsf{K}'\ \mathsf{e}'_i\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'_i\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}'_i : \mathsf{C}_i$, then case analyze.

  1. $\cdot\,\sigma\,\mathsf{K} \vdash \textbf{new}\ \mathsf{C}(\mathsf{e}_1 \ldots \mathsf{e}_n) : \mathsf{C}$
  2. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ by case
  3. $\textbf{class}\ \mathsf{C}\,\{\,\overline{\mathsf{fd}}\ \overline{\mathsf{md}}\,\} \in \mathsf{K}$
  4. $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e}_1 : \mathsf{t}_1 \ldots \cdot\ \sigma\,\mathsf{K} \vdash \mathsf{e}_n : \mathsf{t}_n$
  5. $\overline{\mathsf{fd}} = \mathsf{f}_1 : \mathsf{t}_1 \ldots \mathsf{f}_n : \mathsf{t}_n$ by inversion lemma on (1)
  6. $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{a}_1 : \mathsf{t}_1 \ldots \dot{}\,\sigma\,\mathsf{K} \vdash \mathsf{a}_n : \mathsf{t}_n$ by Inductive hypo on (2)
  7. $\mathsf{K}'\ \textbf{new}\ \mathsf{C}(\bar{\mathsf{a}})\ \sigma$ by Inductive hypo and (6)
  8. $\mathsf{K}' = \mathsf{K}$
  9. $\sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{C}\{\bar{\mathsf{a}}\}]$
  10. a' fresh
  11. $\mathsf{e}' = \mathsf{a}'$ by semantics on (7)
  12. $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{a}' : \mathsf{C}$ by W10 on (8) and (9)
  13. $\mathsf{k} \in \mathsf{K} \implies \cdot\,\mathsf{K} \vdash \mathsf{k}\ \checkmark$ by (2)
  14. $\mathsf{K} \vdash \sigma\ \checkmark$ by premise of WF (2)
  15. $\mathsf{K} \vdash \sigma'\ \checkmark$ by (14), (2), (4)
  16. $\mathsf{K}' \vdash \sigma'\ \checkmark$ by (15), (8)
  17. $\mathsf{K}'\ \mathsf{a}'\ \sigma'\ \checkmark$ by (13), (8), (16), (12)
  18. done by (17) and (12)

- W8
  Apply the IH to the precondition, finding that either e is a value a, e is a stuck state, or $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$, $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$, and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \star$. If it gets stuck at one of the three stuck states, or steps to a new expression, then the case is trivial.
  Consider the a case. Case analyze on t:

  - $\mathsf{t} = \star$. Trivially, $\mathsf{K} <\mathsf{t}>\mathsf{a}\ \sigma \to \mathsf{K}\ \mathsf{a}\ \sigma$.
  - $\mathsf{t} = \mathsf{C}$. Define D such that $\sigma(\mathsf{a}) = \mathsf{D}\{\overline{\mathsf{a}'}\}$. Case analyze on if $\mu\ \mathsf{K} \vdash \mathsf{D} <: \mathsf{C}$.
    - If $\mu\ \mathsf{K} \vdash \mathsf{D} <: \mathsf{C}$, then $\mathsf{K} <\mathsf{t}>\mathsf{a}\ \sigma \to \mathsf{K}\ \mathsf{a}\ \sigma$. Moreover, $\cdot(\mathsf{K}) = \sigma\mathsf{at}$, via W2 and W10.
    - Otherwise, $<\mathsf{t}>\mathsf{a}$ is stuck, and the theorem holds.

- W9
  Trivial, as $\mathsf{e} = \mathsf{a}$.
- W10
  Trivial, as $\mathsf{e} = \mathsf{a}$.

H.6 Type Soundness of KafKa Behavioral Cast

Given that $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ and $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}$, then either there is some e' such that $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$ and $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$ and $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}$ hold, or e is stuck in one of the following forms:

- a
- $\mathcal{E}[\mathsf{a@m(a')}]$
- $\mathcal{E}[<\mathsf{t}'>\ \mathsf{a}]$
- $\mathcal{E}[\prec \mathsf{t}' \succ \mathsf{a}]$

Note that the majority of the proof is identical to that seen in the above proof for soundness of the KafKa core, and is therefore elided. We will only cover the case for the KafKa behavioural cast, WB1.

1. $\cdot\,\sigma\,\mathsf{K} \vdash \mathsf{e} : \mathsf{t}'$ By assumption.
2. $\mathsf{K}\ \mathsf{e}\ \sigma\ \checkmark$ By assumption
3. $\sigma\ \checkmark$ By inversion of (2)
4. Apply IH to (1) and case analyze.

A. Case: **1**. $\mathsf{K}\ \mathsf{e}\ \sigma \to \mathsf{K}'\ \mathsf{e}'\ \sigma'$ **2**. $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$ **3**. $\cdot\,\sigma'\,\mathsf{K}' \vdash \mathsf{e}' : \mathsf{t}'$

  1. $\mathsf{K} \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}\ \sigma \to \mathsf{K}' \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}'\ \sigma'$ By defn. of evaluation environments on (1)
  2. $\cdot\,\sigma'\,\mathsf{K}' \vdash \blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e}' : \mathsf{t}$ By WB1
  3. $\mathsf{K}'\ \mathsf{e}'\ \sigma'\ \checkmark$ By (2,2)

B. Case: e is stuck in one of the allowed stuck states $\mathcal{E}[\mathsf{e}']$

  4. $\blacktriangleleft \mathsf{t} \blacktriangleright \mathsf{e} = \mathcal{E}'[\mathsf{e}']$ By case analysis and defn. of $\mathcal{E}$.
  5. $\mathcal{E}'[\mathsf{e}']$ stuck As e' stuck.

C. Case: $\mathsf{e} = \mathsf{a}$

  6. Let C and $\overline{\mathsf{a}'}$ be such that $\sigma(\mathsf{a}) = \mathsf{C}\{\overline{\mathsf{a}'}\}$
  7. Case analyze on t.

  A. Case: $\mathsf{t} = \mathsf{C}'$

    1. Let D be fresh.
    2. Let a' be fresh.
    3. Let k be $\mathsf{k} = \mathsf{wrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C},\mathsf{K}), \mathsf{mtypes}(\mathsf{C},\mathsf{K}), \mathsf{mtypes}(\mathsf{C}',\mathsf{K}), \mathsf{D})$.
    4. Let $\sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}]$.
    5. $\cdot\,\mathsf{Kk}\,\sigma' \vdash \mathsf{a}' : \mathsf{D}$ W9
    6. $\cdot\ \mathsf{Kk} \vdash \mathsf{D} <: \mathsf{C}'$ correctness of wrap on (3)
    7. $\cdot\,\mathsf{Kk}\,\sigma' \vdash \mathsf{a}' : \mathsf{C}'$ W2 on (6,5)
    8. $\mathsf{K}\ \mathsf{k}\ \mathsf{a}'\ \sigma'\ \checkmark$ WP on (7,3)

  B. Case: $\mathsf{t} = \star$

    9. Let D be fresh.
    10. Let a' be fresh.
    11. Let k be $\mathsf{k} = \mathsf{wrap}(\mathsf{C}, \mathsf{getmds}(\mathsf{C},\mathsf{K}), \mathsf{mtypes}(\mathsf{C},\mathsf{K}), \mathsf{D})$.
    12. Let $\sigma' = \sigma[\mathsf{a}' \mapsto \mathsf{D}\{\mathsf{a}\}]$.
    13. $\cdot\,\mathsf{Kk}\,\sigma' \vdash \mathsf{a}' : \star$ W10
    14. $\mathsf{K}\ \mathsf{k}\ \mathsf{a}'\ \sigma'\ \checkmark$ WP on (13,11)

H.7 Typescript Translation