Monotonic Gradual Typing in a Common Calculus

Benjamin Chung
Northeastern University

Jan Vitek
Northeastern and CTU

ABSTRACT

Gradual typing refers to the notion that programs can be incrementally decorated with type annotations. Languages that support this approach to software development allow for programs being in various states of "typedness" on a scale ranging from entirely untyped to fully statically typed. Points in the middle of this typed-untyped scale create interactions between typed and untyped code, which is where gradual type systems differ. Each gradual type system comes with tradeoffs. Some systems provide strong guarantees at the expense of vastly degraded performance; others do not impact the running time of programs, but they do little to prevent type errors. This paper looks at an intriguing point in the landscape of these systems: the monotonic semantics. The monotonic semantics is designed to reduce the overhead of typed field access through a different enforcement mechanism compared to other gradual type systems. In our previous paper, [1], we described four semantics for gradual typing. This paper uses the framework of that companion paper to present and explore a formulation for the *monotonic* semantics. In comparison to the others, the monotonic semantics is designed to reduce the overhead of typed field access. We translate a common gradually typed source language to a common statically typed target language according to the monotonic semantics, allowing easy comparison to other gradual type systems in our framework.

ACM Reference Format:
Benjamin Chung and Jan Vitek. 2018. Monotonic Gradual Typing in a Common Calculus. In *Proceedings of FTfJP, , July 2018,* 6 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Gradual typing refers to a family of approaches that add types to untyped programs. These approaches each protect the boundaries between typed and untyped code in different ways [4, 6]. Gradual typing allows untyped code to pass values into typed code, and the dynamic values need to be reconciled with the new static types dynamically. The pantheon of gradual type systems arises from different mechanisms to perform this check of types of untyped values.

Object-oriented languages pose a particular problem for gradual typing. It is easy to check if a base value has a type, but problems arise from higher-order behavior (that is, programs that pass around behavior inside values). In a functional language, this manifests when trying to assert an arrow type on a lambda, whose return type cannot be checked at the cast site. Object-oriented languages have this same problem — trying to assert a typed method on an object with an untyped version of the method — but have additional complexity in the form of subtyping.

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

new D().f( <*> new C() )
```

Figure 1: A gradually typed program.

Figure 1 shows a simple example of a gradually typed object-oriented program. Typed expressions in our language use a standard Featherweight Java-like type system with structural subtyping; however, not all expressions in this program are typed. In the surface language, the type ⋆ indicates that a variable is dynamic; thus, it can hold a value of any class; and any method can be invoked, as long as it returns another instance of ⋆ for that variable to receive

The program uses dynamic behavior (a cast) to pass an instance of class C as one of type D in the call to f. C has no guarantee to act like a D, so D-ness must be enforced at runtime. However, deciding the best way to enforce static types on untyped values is an open problem. The program shown here could fail when the instance of C is first passed, fail if f is called on x, or not fail at all if it does not violate the guarantee enforced by the gradual type system.

In a companion paper, to appear at ECOOP 2018 [1], we examined four gradual typing approaches, each of which addresses this problem differently. In that paper, we discussed the optional, transient, behavioral, and concrete semantics. We chose these semantics due to their popularity and relative simplicity; but we elided the monotonic semantics of [7], due to its complexity. This paper places the monotonic semantics into the same framework as the companion paper.

The monotonic semantics sets out to solve a key problem for gradual typing: speed of field access. For example, the behavioral semantics adds a wrapper around every field access which checks the value's type, at the cost of indirection. This can make typed code in a gradually typed language excruciatingly slow [5]. Suppose we have an object $x = \{f : \star = 3\}$, then alias it as $x' =< \{f : \text{int}\} > x$. Despite x''s static type, the field dereference $x'.f$ could return a value of any type. If we run x.f = "hello" at some point, then $x'.f$ is now ill-typed via the reference. As mentioned previously, other semantics deal with this by checking the result of operations in typed code, lazily enforcing soundness of type annotations.

We would like to thank Celeste Hollenbeck for assistance with the writing of this paper.

FTfJP, July 2018,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The monotonic semantics is an *eager* mechanism to enforce types, checking values assigned to fields on assignment rather than dereference. Using the monotonic semantics, once a value in the heap is typed, it will always remain of that type. In the example above, once we aliased x as x′, x′′'s type (e.g. that f has type int) will be enforced upon usages of x. Now, our assignment x.f = "hello" will fail, because we are assigning a value of improper type to the field f of the object. Monotonic uses this mechanism to guarantee that all values referred to by typed references will be of the correct type.

In this paper, we present a formalization of the monotonic semantics in the same framework used in our companion paper [1] for comparing gradual type systems. We use a common gradually-typed object-oriented source language and translate it to a common target language (called KafKa) according to the monotonic semantics. We then use this formalization to discuss the monotonic semantic properties, and compare it to the gradual type systems presented in our companion paper.

2 MONOTONIC GRADUAL TYPES

The idea behind the monotonic semantics' was originally put forward by Siek and Vitousek in [7] for their Reticulated Python language implementation. Kuhlenschmidt, Almahallawi, and Siek later implemented monotonic again as part of their Grift functional programming language [2], with the goal of demonstrating its performance. For similar purposes, it was again implemented for TypeScript by Richards, Arteca, and Turcotte [3], leveraging the correspondance between monotonic's concept of precision and VM object shapes to improve performance.

The core tenet of the monotonic semantics is that types only ever get more precise. This design gives rise to its name, as types of values in a monotonic language's heap monotonically get more static. In an object-oriented language, each object in the heap has an associated type, as indicated by its run-time tag. This tag identifies the class of the object, and it is used for dispatching method invocations. When a reference to an object is cast to a more precise type, intuitively a type with fewer occurrences of ⋆, that new type is attached to the value in the heap. A type error will occur if untyped code later tries to read fields, write fields, or call methods in a way that violates the updated type. For example, an object may start out under type {m(⋆): ⋆}, then become more static when cast to {m(int): int}. This new type will be stored in the heap and enforced on untyped callers. By locking in applied types forevermore, typed code under the monotonic semantics can assume that mutable state will remain well-typed.

Through this mechanism, the monotonic semantics is able to provide a stronger guarantee about values inhabiting mutable typed values when compared to other gradual type systems. Unlike the optional semantics, the monotonic semantics provides a soundness guarantee. It also allows the elision of guards at typed dereference sites, unlike the transient and behavioral semantics. Finally, the monotonic guarantee is less strict at the original cast site than the concrete semantics. Additional details about these other semantics can be found in our companion paper [1].

As a result of this stronger soundness guarantee, the monotonic system has the potential to provide substantial benefits to the performance of gradually typed languages. Variables are guaranteed to refer to values of their statically known type, and therefore their types can be used to discover the memory layout of the underlying object. Therefore, the objects' memory layout is known statically and field references made direct. This optimization has the potential to reduce one of gradual typing's largest overheads.

In the context of object orientation, monotonic's ability to protect mutable state without the need for wrappers is a significant advantage. Other gradual type systems need guards, wrappers, or a stronger notion of soundness in order to guarantee safety of field access. The monotonic semantics, in contrast, prevents untyped code from assigning ill-typed values to previously typed references, allowing unchecked dereferencing in typed code.

3 FORMALISM

We use the framework we set out in our companion paper to provide a simple presentation of the monotonic semantics in an object-oriented setting. Instead of using formal languages specially crafted to support each semantics, we use a common source and target language for every gradual typing semantics. The semantics is then encoded into the translation, allowing comparison of gradual type systems by examining their translation.

3.1 Source Language

Our common source language is similar to Featherweight Java (FJ), with a few changes. This language has classes but relies on structural subtyping instead of the nominal subtyping adopted by FJ. For our purposes, there is no need for inheritance. Structural subtyping is needed to support the evolution of types, as casts are being applied. The source language also introduces a dynamic type, ⋆, and a convertibility relation, allowing gradually typed programs to be expressed. This same source language and source language type system is shared between the translation presented here and the four in our main paper, allowing the same programs to be run under all of our semantics. We replicate the static semantics for the source language here, in Figure 2. The dynamic semantics of the source language is given by a translation to a statically-typed language, which we will present next.

Programs in the source language consist of a class table, K, and an initial expression, e. Classes consist of field and method definitions. Fields have a name and a type, f:t. Method definitions have names, types for their single argument and return type, and an expression. Expressions consist of variables, the self-reference this, field read and write this.f and this.f = e, invocation e.m(e), and new object creation **new** C(e_1..). Each class has an implicit default constructor that merely assigns arguments to fields. All operations conform to what they would do in Featherweight Java, and the type system is straightforward.

The static type system allows the implicit conversion of typed and untyped terms via the convertibility operator, written $K \vdash_s t \Longmapsto t'$ – type t is convertible to type t′. The relation is used both for up-casting and for conversions of ⋆ to non-⋆ types. SUB allows transparent up-casting, while TOA and ANYC allow implicit conversion to and from the dynamic type. To avoid collapsing the type

hierarchy, convertibility is not transitive. With a naive semantics, uses of TOA would break soundness, since dynamically typed values would enter statically typed code without any kind of guard.

The source language enforces field privacy. Fields can only be accessed from an object's own methods. Furthermore, fields do not show up in type signatures. Subtyping between objects is computed solely on methods signatures.

3.2 KafKa

KafKa's design follows the surface language with some differences. Expressions are extended with that, a distinguished variable used for wrapped object reference, typed and untyped method invocations, subtype and monotonic casts. Evaluation is likewise standard with a few exceptions, with an evaluation context consisting of a class table K, expression being evaluated e, and heap σ, mapping from addresses a to objects, denoted C{a . . .}. Due to the need for dynamically generated classes, the class table K is part of the state, and not the evaluation context. KafKa has two invocation forms, the untyped invocation $e@m_{\star\to\star}(e')$ and the typed invocation $e.m_{C\to D}(e')$, both denoting a method call to method m with argument e'. This is unusual for two reasons. First, most gradual type systems have no explicit untyped invocation form, and second, the typed form also specifies the argument type and return type. Our system explicitly indicates which function calls are statically guaranteed to succeed and which must be dynamically checked. It also allows for a limited form of overloading, where both typed and untyped versions of a method may share a name.

We use a gray background to indicate changes to KafKa in figure 2 to support monotonic. The only addition is that of the monotonic cast. The next section focuses on giving a semantics to that cast.

4 MONOTONIC SEMANTICS

The monotonic semantics is defined by a translation from the surface language down to KafKa and the meaning of the monotonic cast.

The translation is syntax directed, each surface class is translated to an eponymous KafKa class. Types of methods and fields are retained in the translation. Expressions are translated with $M[\![e]\!]_\Gamma$. Variables are translated to themselves. Calls are translated to either dynamic calls or static calls depending on the type of the receiver. If the receiver is of type $\star$, then the argument must be of type $\star$. If the receiver of some class C then the argument must be of type D, where D is the expected argument type of the method. We use a variant of the translation relation that performs a cast if necessary on its argument expression, $M(\!|e|\!)^t_\Gamma$. The inserted cast is the monotonic cast, $\triangleleft t \triangleright e'$.

The semantics of monotonic arises from the implementation of the monocast metafunction, used in the evaluation of monotonic casts shown in figure 7. At a high level, the cast operation computes the meet of the source types (the most specific join of them), then applies it by replacing the class associated with the object in memory. This is depicted in rule CM.

The first step in this operation is the meet operator. The meet operation is akin to the one over a lattice, taking two types and always producing a type higher on the type lattice. Two types have a meet if they are structurally the same up to $\star$ types distributed throughout their structure. The meet itself consists of replacing all $\star$ types in both types with the most specific known type in that position from either type.

For example, if we were to compute the meet between C and D as shown in figure 4, we would end up with the type E. The $\star$ typed argument to m in C is taken to be of type D from class D, while the converse is used for n in D. The class E then is more specific than both C and D.

This operation relies on the meet metafunction which takes the two target types, a context (for use in recursive types), and the class table. It then outputs a new class, the meet of the two types, and an updated classes table (meet can create multiple types). The context P, either empty $\cdot$ or a context extended with a mapping $P\ (C, D) \mapsto C'$, is used to ensure termination of the meet operation on recursive types by storing the names of meet-generated classes. Meet is deterministic, so meet of previously-encountered types will be identical. We similarly extend meet to operate over method definitions, picking out the comparable method definition in the target method, and generate a suitable method. When two list of methods are met, their result is passed into ifcgen, which generates an "interface". An interface is trivial class that exists only for the purpose of giving us a type signature. It will never be used. If we were to use it we would run into trouble. Suppose we start out with $x = \textbf{new}\ C()$, then alias it with $x' = \triangleleft D \triangleright x$. If we then call $x@m_{\star\to\star}(\textbf{new}\ D())$, and the underlying object is an instance of E, the dynamic call will fail as no dynamically-typed method exists. As a result of this, our system does not deem E to be in a subtyping relationship with either C or D. Moreover, ifcgen makes no attempt to produce correct method bodies.

To fix this, we need to add additional guard methods that ensure the presence of an untyped target method even if cast. The meet of classes C and D in our system is therefore class F as shown in figure 5, which has both the typed and untyped methods specified in both C and D. Class F is a proper structural subtype of both C and D, and its methods perform semantically correct operations. It is generated by the wrap metafunction, which produces an equivalent class from source and target classes. It ensures that any untyped method in the source is retained in the target, and that typed methods are inserted.

5 EXAMPLE

To illustrate the operation of our semantics, we present a fully worked example of a simple program running under the monotonic semantics. This program, shown in figure 8, is first translated from source to KafKa and then executed according to the KafKa semantics described above.

This program is designed to illustrate the cast semantics of monotonic. The method alias aliases its argument, casting it to both C and D. The value cast to C is then retrieved and its method check is then called (in untyped code) with an I. The reference is identical to the one originally passed to alias and no wrapper has been applied. One would expect then that the untyped call to check would pass, since the body of check is the identity function, but under the monotonic semantics it fails.

$$e ::= x \mid \text{this} \mid \text{this.f} \mid \text{this.f} = e \mid e.m(e) \mid \textbf{new}\ C(e_1..)$$
$$k ::= \textbf{class}\ C\ \{\ fd_1..\ md_1..\ \} \qquad md ::= m(x\colon t)\colon t\ \{e\} \qquad fd ::= f\colon t \qquad t ::= \star \mid C$$

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

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

$$\frac{f_1\colon t_1.. \in K(C) \quad \Gamma\ K \vdash_s e_1 : t_1'.. \quad K \vdash_s t_1' \Rightarrow t_1..}{\Gamma\ K \vdash_s \textbf{new}\ C(e_1..) : C}$$

$$\text{SUB}\ \frac{\emptyset\ K \vdash_s t <: t'}{K \vdash_s t \Rightarrow t'} \qquad \text{TOA}\ \frac{}{K \vdash_s t \Rightarrow \star} \qquad \text{ANYC}\ \frac{}{K \vdash_s \star \Rightarrow t}$$

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

$$e ::= x \mid \text{this} \mid \text{that} \mid \text{this.f} \mid \text{this.f} = e \mid e.m_{t\to t}(e) \mid \textbf{new}\ C(e_1..) \mid e@m_{\star\to\star}(e) \mid a \mid a.f \mid a.f = e \mid \triangleleft t \triangleright e$$
$$k ::= \textbf{class}\ C\ \{\ fd_1..\ md_1..\ \} \qquad md ::= m(x\colon t)\colon t\ \{e\} \qquad fd ::= f\colon t \qquad t ::= \star \mid C$$
$$E ::= a.f = E \mid a.m_{t\to t}(E) \mid a@m_{\star\to\star}(E) \mid \textbf{new}\ C(a_1..\ E\ e_1..) \mid E.m_{t\to t}(e) \mid E@m_{\star\to\star}(e) \mid \triangleleft t \triangleright E \mid \square$$

$$\text{KT-VAR}\ \frac{\Gamma(x) = t}{\Gamma\ \sigma\ K \vdash x : t} \qquad \text{KT-SUB}\ \frac{\Gamma\ \sigma\ K \vdash e : t' \quad \cdot\ K \vdash t' <: t}{\Gamma\ \sigma\ K \vdash e : t} \qquad \text{KT-READ}\ \frac{\Gamma(\text{this}) = C \quad f\colon t \in K(C)}{\Gamma\ \sigma\ K \vdash \text{this.f} : t}$$

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

$$\text{KT-CALL}\ \frac{\Gamma\ \sigma\ K \vdash e : C \quad \Gamma\ \sigma\ K \vdash e' : t \quad m(t)\colon t' \in K(C)}{\Gamma\ \sigma\ K \vdash e.m_{t\to t'}(e') : t'} \qquad \text{KT-DYNCALL}\ \frac{\Gamma\ \sigma\ K \vdash e : \star \quad \Gamma\ \sigma\ K \vdash e' : \star}{\Gamma\ \sigma\ K \vdash e@m_{\star\to\star}(e') : \star}$$

$$\text{KT-NEW}\ \frac{\Gamma\ \sigma\ K \vdash e_1 : t_1.. \quad \textbf{class}\ C\ \{\ f_1\colon t_1..\ md_1..\ \} \in K}{\Gamma\ \sigma\ K \vdash \textbf{new}\ C(e_1..) : C} \qquad \text{KT-MONOCAST}\ \frac{\Gamma\ \sigma\ K \vdash e : t'}{\Gamma\ \sigma\ K \vdash \triangleleft t \triangleright e : t} \qquad \text{KT-REFTYPE}\ \frac{\sigma(a) = C\{a'_1..\}}{\Gamma\ \sigma\ K \vdash a : C}$$

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

$$\begin{array}{llllllll}
K & \textbf{new}\ C(a_1..) & \sigma & \to & K & a' & \sigma' & \textbf{where}\ a'\ \text{fresh}\quad \sigma'{=}\sigma[a' \mapsto C\{a_1..\}] \\
K & a.f_i & \sigma & \to & K & a_i & \sigma & \textbf{where}\ \sigma(a) = C\{a_1, ..a_i, a_n..\} \\
K & a.f_i = a' & \sigma & \to & K & a' & \sigma' & \textbf{where}\ \sigma(a) = C\{a_1, ..a_i, a_n..\}\quad \sigma'{=}\sigma[a \mapsto C\{a_1, ..a', a_n..\}] \\
K & a.m_{t\to t'}(a') & \sigma & \to & K & e' & \sigma & \textbf{where}\ e' = [a/\text{this}\ a'/x]e\quad m(x\colon t_1)\colon t_2\ \{e\} \in K(C)\quad \sigma(a) = C\{a_1..\}\quad \emptyset\ K \vdash t <: t_1\quad \emptyset\ K \vdash t_2 <: t' \\
K & a@m_{\star\to\star}(a') & \sigma & \to & K & e' & \sigma & \textbf{where}\ e' = [a/\text{this}\ a'/x]e\quad m(x\colon \star)\colon \star\ \{e\} \in K(C)\quad \sigma(a) = C\{a_1..\} \\
K & \triangleleft t \triangleright a & \sigma & \to & K' & a' & \sigma' & \textbf{where}\ \text{moncast}(a, t, \sigma, K) = K'\ a'\ \sigma' \\
K & E[e] & \sigma & \to & K' & E[e'] & \sigma' & \textbf{where}\ K\ e\ \sigma \to K'\ e'\ \sigma'
\end{array}$$

Figure 3: KafKa **syntax, static and dynamic semantics, and monotonic translation.**

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

Figure 4: Meet C and D

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

Figure 5: wrap result

Before explaining the semantics of this program in more detail, we will first describe the top level type-driven transformation. First,

$$
\begin{array}{rll}
M[\![x]\!]_\Gamma = x & & \\
M[\![e_1.m(e_2)]\!]_\Gamma = e_1'@m_{\star\to\star}(e_2') & & \textbf{if } K,\Gamma \vdash e_1 : \star \ \wedge\ M[\![e_1]\!]_\Gamma = e_1' \ \wedge\ M(\!|e_2|\!)^{\star}_{\Gamma} = e_2' \\
M[\![e_1.m(e_2)]\!]_\Gamma = e_1'.m_{D_1\to D_2}(e_2') & & \textbf{if } K,\Gamma \vdash e_1 : C \ \wedge\ m(D_1) : D_2 \in K(C) \ \wedge\ M[\![e_1]\!]_\Gamma = e_1' \ \wedge\ M(\!|e_2|\!)^{D_1}_{\Gamma} = e_2' \\
M[\![\textbf{new } C(e_1..)]\!]_\Gamma = \textbf{new } C(e_1'..) & & \textbf{if } f_1 : t_1 \in K(C) \ \wedge\ e_1' = M(\!|e_1|\!)^{t_1}_{\Gamma} \ .. \\
M(\!|e|\!)^{t}_{\Gamma} = e' & & \textbf{if } K,\Gamma \vdash e : t' \ \wedge\ K \vdash t' <: t \\
M(\!|e|\!)^{t}_{\Gamma} = \triangleleft t \triangleright e' & & \textbf{if } K,\Gamma \vdash e : t' \ \wedge\ K \vdash t' \not<: t
\end{array}
$$

Figure 6: Monotonic Translation

$$
\text{CM}\ \frac{\sigma(a) = C\{a'..\} \quad C'' \text{ fresh} \quad md.. = meths(C,K) \quad mt.. = mtypes(C,K) \quad mt'.. = mtypes(C',K) \quad meet(C,t,\cdot,K) = C'\ K' \quad K'' = wrap(C,md..,mt..,mt'..,C'',K)\ K' \quad \sigma' = \sigma[a \mapsto C''\{a'..\}]}{moncast(a,t,\sigma,K) = K''\ \sigma'}
$$

$$
\text{TM1}\ \frac{}{meet(C,\star,P,K) = C\ K} \qquad \text{TM2}\ \frac{}{meet(\star,C,P,K) = C\ K} \qquad \text{TM3}\ \frac{}{meet(t,t,P,K) = t\ K} \qquad \text{TM5}\ \frac{P(C,D) = C'}{meet(C,D,P,K) = C'\ K}
$$

$$
\text{TM4}\ \frac{(C,D) \notin P \quad P' = P\ (C,D) \mapsto C' \quad \begin{array}{c} C' \text{ fresh} \\ meet(mtypes(C,K), mtypes(D,K), P', K) = mt''..\ K' \end{array} \quad K'' = K'\ ifcgen(mt''..,C')}{meet(C,D,P,K) = C'\ K''}
$$

$$
\text{MM1}\ meet(mt..,\cdot,\Gamma,K) = mt..\ K \qquad\qquad \text{MM2}\ meet(\cdot,mt..,\Gamma,K) = mt..\ K
$$

$$
\text{MM6}\ \frac{m(t_3) : t_4 \in mt_2 \quad meet(t_3,t_1,\Gamma,K) = t_5\ K' \quad meet(t_2,t_4,\Gamma,K') = t_6\ K'' \quad meet(mt_1..,mt_2..,\Gamma,K') = mt_3..\ K''}{meet(m(t_1) : t_2\ mt_1..,mt_2..,\Gamma,K) = m(t_5) : t_6\ mt_3..\ K''}
$$

$$
\begin{array}{l}
ifcgen(mt..,D) = \\
\quad \textbf{class } D\ \{ \\
\qquad m(x:t) : t'\ \{\triangleleft t' \triangleright x\} \qquad \forall\ m\ .\ m(t) : t' \in mt.. \\
\quad \}
\end{array}
$$

$$
\begin{array}{ll}
wrap(C,md..,mt..,mt'..,D,K) = & \\
\quad \textbf{class } D\ \{ & \\
\qquad f : t & \forall\ f : t \in fields(K,C)\ . \\
\qquad m(x:\star) : \star\ \{\ \triangleleft\ \star \triangleright this.m_{t_1\to t_2}(\triangleleft t_1 \triangleright x)\ \} & \forall\ m\ .\ m(x:\star) : \star\ \{e\} \in md..\ \wedge\ m(C_1) : C_2 \in mt'.. \\
\qquad m(x:C_1) : C_2\ \{\triangleleft C_2 \triangleright [(\triangleleft\ \star \triangleright x)/x]e\ \} & \forall\ m\ .\ m(x:\star) : \star\ \{e\} \in md..\ \wedge\ m(C_1) : C_2 \in mt'.. \\
\qquad m(x:C_1) : C_2\ \{e\} & \forall\ m\ .\ m(x:C_1) : C_2\ \{e\} \in md..\ \wedge\ m(C_1) : C_2 \in mt'.. \\
\qquad m(x:\star) : \star\ \{e\} & \forall\ m\ .\ m(x:\star) : \star\ \{e\} \in md..\ \wedge\ m(C_1) : C_2 \notin mt'.. \\
\quad \}
\end{array}
$$

Figure 7: Monotonic Cast Operations

casts are inserted wherever a statically typed value is passed to the dynamic type $\star$ and vice versa. Then, typed methods have untyped shims generated, so they can be called in a protected manner from untyped code.

Execution starts with the expression `new E().alias(<*>new C()).check(<*>new C()).` `new E()` evaluates to a reference a to an object E{}, upon which alias is invoked with argument **new** C() (which evaluates to a_1, mapping to C{}). alias then expands to $\triangleleft \star \triangleright$**new** T($\triangleleft$C$\triangleright$ a_1, $\triangleleft$D$\triangleright$ a_1). The first $\triangleleft$C$\triangleright$ a_1 is a no-op, as a_1 is already of type C, but the second $\triangleleft$D$\triangleright$ a_1 modifies the runtime type of the value stored at a_1.

D defines check to take and return instances of I, while C has check taking and returning values of type $\star$. As D's definition is more precise, it will take priority. As a result, the result of meet will be C′, with **class** C′ { check(x: I): I {x} } appended to the class table. monWrap is then applied, producing the final new class C″ from C′, consisting of **class** C″ { check(x: I): I {$\triangleleft$I$\triangleright$ $\triangleleft$ $\star\triangleright$x} check(x: $\star$): $\star$ {$\triangleleft$ $\star\triangleright$this.check$_{\mathsf{I}\to\mathsf{I}}$($\triangleleft$I$\triangleright$ x)} }. This is then used to overrite a_1's class in the heap. a_1 originally referred to C{}, but it will now refer to C″{} after the cast.

At this point, the initial expression has been reduced to $\triangleleft \star \triangleright$**new** T($a_1$, a_1).first$_{\star\to\star}$($\star$)@check$_{\star\to\star}$($\triangleleft \star \triangleright$**new** C()). The inner expression **new** T(a_1, a_1).first$_{\star\to\star}$(a_1) evaluates to a_2.first$_{\star\to\star}$(a_1) where a_2 refers to T{a_1, a_1} in the heap. The first function then is called, returning a_1 under type $\star$.

This leaves us with a_1@check$_{\star\to\star}$($\triangleleft \star \triangleright$**new** C()). We evaluate **new** C() to a_3 (mapped to C{}), and make the invocation. A method check under type $\star$ and $\star$ exists, so the program does not get stuck. a″ is an instance of C″, so it is dispatched to the generated method check made by the monocast operation. As a result, our new term is $\triangleleft \star \triangleright a_1$.check$_{\mathsf{I}\to\mathsf{I}}$($\triangleleft$I$\triangleright$ a_3). However, at this point, we get stuck, as no meet exists between the runtime type of a_3 (mapped to C{}) and I.

The example illustrates an interesting property of the monotonic semantics. Despite this program working perfectly in an untyped language, the monotonic semantics caused it to fail. More interestingly, no other gradual typing semantics would experience a cast error on this program. This is because none will retain the type cast of the first alias to a_3 in the underlying object, since the alias is immediately thrown away.

```
class C{ check(x:*) {x}}
class D{ check(x:I):I {x}}

class I{ dif(x:*):* {x}}

class T {
 f1:C f2:D
 first(x:*):* {this.f1}}
class E {
 alias(x:*):* {
  new T(x,x).first(x) }}


new E().alias(new C())
       .check(new C())
```

Source

```
class C { check(x:*) {x}}
class D {
 check(x:I):I {x}
 check(x:*):* {
    <*>this.m(<I>x) }}
class I { dif(x:*):* {x}}

class T {
 f1:C f2:D
 first(x:*):* {
    <*>this.f1}}
class E {
 alias(x:*):* {
    <*>new T(<C>x,<D>x)
             .first(x)}}

new E().alias(<*>new C())
       @check(<*>new C())
```

KafKa

Figure 8: Monotonic Semantics Example

6 CONCLUSION

This paper has presented the monotonic semantics for gradual types of object oriented programs as a translation from source programs to KafKa. One of the key properties of that translation is that the monotonic casts attempts to compute the meet of the class of the target object and the requested type. If the two cannot be reconcilled the cast fails, otherwise the type of the object is updated in place to reflect the new type. This allows monotonic programs to avoid the potentially unbounded layers of wrappers of previous approaches. The monotonicity property ensure that there can be at most as many wrappers as there are occurrences of $\star$ in the type. Practical implementation can have a single wrapper.

One simplification that we have made here comes from the design of KafKa. KafKa does not reflect fields in the types of objects, they are fully encapsulated. This means that when computing the meet we do not have to update the type of fields. A language that makes fields accessible outside of objects would have to also meet the values referenced by fields, which may mean that a meet operation could update multiple objects.

Given the complexity of the operations performed by meet, more experience is needed to determine if the approach is viable. The implementation of Monotonic Reticulated Python has not yet been widely adopted, so few programs have been written in this style.

REFERENCES

[1] Benjamin Chung, Paley Li, Francesco Zappa Nardelli, and Jan Vitek. 2018. KafKa: Gradual Typing for Objects. In *European Conference on Object-Oriented Programming (ECOOP)*.

[2] Andre Kuhlenschmidt, Deyaaeldeen Almahallawi, and Jeremy G. Siek. 2018. Efficient Gradual Typing. *CoRR* abs/1802.06375 (2018). arXiv:1802.06375 http://arxiv.org/abs/1802.06375

[3] Gregor Richards, Ellen Arteca, and Alexi Turcotte. 2017. The VM Already Knew That: Leveraging Compile-time Knowledge to Optimize Gradual Typing. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 55 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133879

[4] Jeremy Siek. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*. http://ecee.colorado.edu/~siek/pubs/pubs/2006/siek06_gradual.pdf.

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

[6] Sam Tobin-Hochstadt and Matthias Felleisen. 2006. Interlanguage migration: from scripts to programs. In *Symposium on Dynamic languages (DLS)*. https://doi.org/10.1145/1176617.1176755

[7] Michael Vitousek, Andrew Kent, Jeremy Siek, and Jim Baker. 2014. Design and Evaluation of Gradual Typing for Python. In *Symposium on Dynamic languages (DLS)*. https://doi.org/10.1145/2661088.2661101