#58 Gradual Types for Objects Redux

Email notification
Select to receive email on updates to reviews and comments.

PC conflicts
Francesco Zappa Nardelli
Frank Tip
Jan Vitek
Tobias Wrigstad

Submitted

Submission 14 Jan 2017 6:50:30am EST · 66becc15

▸ **Abstract**

The enduring popularity of dynamically typed languages has given rise to a cottage

[more]

▸ **Authors**

B. Chung, P. Li, J. Vitek [details]

	OveMer	RevExp
Review #58A	C	X
Review #58B	C	Y
Review #58C	D	Y
Review #58D	C	Y
Review #58E	D	X

You are an **author** of this submission.

Edit submission · Add response

Reviews in plain text

Review #58A

Plain text

Overall merit
C. Weak reject

Reviewer expertise
X. I am an expert in this area

Paper summary
The paper reviews five existing approaches to gradual typing for object-oriented languages: TypeScript, Thorn, Transient Python, Typed Racket, and Monotonic Python. Each of them is modeled using a core calculus, KafKa, with mutable state, dynamic generation of wrapper classes, and different kinds of casts. Quoting from the conclusion, this "offers the opportunity for comparing and contrasting these gradual typing systems within a[n] unified framework" and "highlights the essences which makes each system unique".

Points For and Against

+ Provides a nice overview of recent work on gradual typing.

– No new results.

Comments for author
According to the introduction, the aim of the paper is "to expose some of the forces that have influenced existing systems and discuss the implications of key design decision" and "to lay the agenda for future investigations". However, for someone already familiar with the literature in this area, there are no new interesting observations, nor any suggestion for an agenda for future work.

This might be a nice journal (survey) paper, but it does not seem appropriate for ECOOP.

Minor remarks:

– "One outcome of our work is showing how all of the gradual type systems are observationally distinct". That is hardly surprising to anyone.

– The title and the introduction of Section 5 are misleading: the section is not about *translating* but about *modeling* (as it abstracts away from many aspects of each of the source languages while focusing on parts related to gradual typing that make each language different from the others).

- Can you provide a source to the "anecdotal evidence" mentioned on pages 4-5 that blame tracking is helpful to programmers?

- "Gradual typing is no longer simply a popular research topic in academia" (first statement in the Conclusion section). That certainly depends on whether you mean "gradual typing" with or without blame tracking (that is, in the sense of Siek et al., Refined Criteria for Gradual Typing, SNAPL 2015, or in the sense of "optional typing" à la Bracha and others). In the former case, the statement is incorrect (as evident from Figure 1). It would be fair to the readers to point out that gradual typing with blame tracking has *not* gone beyond academic research, even though that observation may not fit well with the paper's strong focus on techniques for blame tracking.

Review #58B

Plain text

Overall merit

C. Weak reject

Reviewer expertise

Y. I am knowledgeable in this area, but not an expert

Paper summary

The paper takes a systematic approach to the comparison of gradual typing implementations, translating 5 prominent gradual typing approaches to a single core calculus with a variety of casts. Forms of TypeScript, Thorn, Transient Python, Typed Racket, and Monotonic Python are translated to KafKa, a new core language including four types of casts: subtype, shallow, behavioral, and monotonic. This makes precise the distinctions in semantics between a range of approaches, allowing them to be precisely compared. The paper also includes a set of litmus test programs (with equivalents in each source type system) whose runtime semantics highlight the differences between the cast semantics.

Points For and Against

Points in favor:

+ The paper targets an important problem whose results will be very useful
+ The paper takes the right general approach to comparing these systems
+ The paper clearly explains the design space, and is mostly self-contained despite covering a lot of ground

Points against:

- The presentation of monotonic casts was not very clear
- Much of the technical machinery is treated inconsistently, adding confusion
- The generative casts share little in common with the lightweight casts, making the value of having them reside in a common core language (as opposed to a common framework) harder to see
- The monotonic casts are modeled in a very different way from prior work (wrapping instead of updating), which seems to complicate the model but isn't well-motivated

Comments for author

The paper takes the right general approach to contrasting the semantics, and is indeed the first paper to lay out the semantics of all of these systems in a common formulation --- this adds quite a bit of clarity without the requirement of so much study of the formulated systems. The relationship between the systems is spelled out quite clearly (aside from my concerns about the generative casts), and even identifying the placement of casts by each approach in the translation to KafKa is very useful.

My biggest concern with the paper is that the treatment of monotonic casts appears to be poorly explained and deviates in major ways from prior implementations [9,21] without good reason. The explanation is sufficiently poor that I needed to stare at tmeet and the appendix for a long time to figure out how it was possible for a monotonic cast to fail in the semantics.

The lack of validation against implementations or other formalizations is also problematic. This is important in general, but especially when monotonic casts are implemented so differently here from other work.

Unclear discussion of monotonic casts

Section 5.6, Figure 14, and Section 5.7 are not clear, due to a combination of confusing writing and inconsistencies in the technical presentation.

Figure 14 shows several different arities for the synthetic cast insertion judgment. Looking at what appears to the right of the $\Uparrow$, which for lack of a better term I'll refer to as something the judgment "produces":

- MOS1 "produces" a type
- MOS2 "produces" a type and two class tables
- MOS5 "produces" a type, a class table, and a class definition
- MOS2 and MOS3, in the hypotheses that recursively refer to the synthetic judgment, assume it produces a type and one class table

I'm guessing MOS1 should produce a class table as well, and in the other cases the judgment should be read as producing a type and single class table (with some rules concatenating tables in the conclusion, without indicating so explicitly). Please tell me if I'm mistaken. This is a switch in judgment form from the other translations, and was not introduced or explained (note that the translation rules for Typed Racket didn't actually appear in the main paper).

The paper contains other passages that seem intended to provide intuition, but actually say things that appear subtly incorrect. For example, on page 20, the paper remarks that "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," but this suggests that casting an object of type (modulo class names, etc.) $(A,\star)$ to $(\star,A)$ will do nothing instead of refining the type to (A,A). Siek et al. [21] use a notion of precision to guide refinement, which is concerned with producing more precise types. I suspect you mean to suggest that the casts will only narrow dynamic types to concrete types, regardless of quantity.

Why not in-place updates for monotonic casts?

The motivation for modeling monotonic casts by wrapping rather

than in-place update was not explained. On page 20, the paper claims "The monotonic cast semantics in KafKa cannot be implemented using the in-place replacement technique [21]." Why not? The paper is not absolutely clear on it, but that paragraph seems to suggest this is because KafKa doesn't permit a field and getter/setter methods of the same name, as method invocation and field access are overloads of the same syntactic construct. This would be a concern because the casts on field sets are implemented via setter-method wrappers.

This seems like an artificial problem. On one hand, wouldn't this be better solved by changing KafKa to use more explicitly use something akin to C#'s default implementations of properties (https://msdn.microsoft.com/en-us/library/bb384054.aspx) so the getters/setters can be overridden when an object's class is changed, without changing the rest of the object representation? This seems like a much simpler than the added complexity of inserting the extra wrappers (which has the extra cost of breaking reference identity when it's not necessary). On the other hand, why would it be an issue to update the object to replace a physical field with setters/getters? Field reads and getter invocation are syntactically identical, so it seems like switching the object representation wouldn't cause problems. Or why not extend the dynamic semantics to check for a required cast to perform? You could surely arrange things such that the check never did anything for objects that had not been monotonically cast.

On its own this choice is not that important, but the paper's goals are to clearly explain the semantics of *existing* approaches. While this wrapper-based coercion may correctly model the semantics of monotonic casts, having the semantics be destructive as in other work [9,21] would seem to better meet the paper's top-level objectives. Moreover, monotonic casts preserve object identity in prior work, while here the object identity changes. This isn't observable in the core language here, but if future work needed that preserved, it couldn't build (directly) on this formalization.

Poor/No explanation of cast failure

The paper never actually explains the cases where monotonic casts should fail. Looking at Appendix H, the only case I see the cast getting stuck is if two types that are being joined have incomparable method name sets (i.e., each has a method the other lacks), in which case tmeet will be undefined. This makes sense for structural meets between recursive types, but is not explained in the paper. However, in the case that the meet exists but the object being coerced lacks a method present in one of the types (but contains all methods of the other), I have a hard time seeing where that will fail.

Lesser comments

- page 10: In the type rule for statically dispatched method calls, wouldn't we prefer to bias the method lookup towards typed method signatures? Section 4.2 says "typed methods are invoked with static calls and untyped methods are invoked with dynamically resolved calls." The current type rule would seem to permit static calls to match the type signature of the untyped method as well as the typed method, but it sounds like the dynamic semantics are actually stricter. If the dynamic semantics don't let static calls dispatch to untyped methods when both are present, then the static dispatch type rule is unsound.
- page 11: I have no idea how to parse this box near the top of the page. It seems like it's supposed to be a graphically-suggestive presentation of some conditions on the translation, but I don't get it.
- Figure 4: These rules could really use paretheses around the compound expressions, particularly the casts, to make the figure easier to parse.
- Figure 4 / Section 4: The behavioral and monotone casts are omitted from the initial formal presentation, and added later. You should mention this explicitly in Section 4.
- Figure 5 (and others): I'm not sure what the translation of dot to dot is for in TSC2 (other translations have similar rules). Is this supposed to be translating the hole for an evaluation context? If so, why is that there, I thought the translations worked on complete programs?
- Section 5.4: This first sentence about the "macro model" makes no sense to me, as I'm not an expert on Typed Racket. It could use a brief explanation.
- Figure 9: wrap is invoked with two different arities here: 5 arguments in BC1, 4 arguments in BC2. Is there a missing argument in BC2?
- Figure 10: I *think* Figure 10 is typeset immediately below an inline example of wrap. But visually it looks like both wrap

examples are part of the figure, making it hard to interpret the reference to Figure 10 in the "Copying the behaviour..." paragraph. I know this is partly due to luck of the layout algorithm, but consider playing with the optional placement arguments to the figure environment.
- The explanation of Figure 12 on page 16 is unclear. Scrutinizing the figure, it seems like the key change is that *calls* to lifted methods also acquire some casts, rather than just the entry and exit of each method. But that change isn't spelled out. And if I'm wrong about the type of change required, then it really needs to be spelled out. As written, that paragraph simply doesn't explain the details of inserting casts inside the method bodies.
- page 22: The first two sentences on this page (one of which

  continues from the previous page) seem to contradict each other.

Grammar nitpicks

Please have someone review your paper for grammar issues. While the paper was not harder to read for it, the paper contains many grammatical errors, ranging from incorrect pluralization and verb conjugation to spelling and repeatedly misusing "effect" for "affect." Below are some of the ones I noticed, and a few places the writing could be slightly smoother.

- page 2: "the object references by a" should probably be "referenced"
- page 3: "type annotations to not effect performance." "effect" used as a verb requires an object (i.e., it effects *a particular change*). Without a specific influence, the correct word here is "affect." This happens elsewhere, too.
- page 3: A quibble: It seems odd to say "C# 4.0 adds" (present tense) for a system that was released 7 years ago. Similarly mildly awkward to say Thorn "extend[s]" the C# approach, when Thorn is older than C# 4.0. The paragraph mixes temporal connotations where you're really explaining how conceptual components are assembled.
- page 3: "a strongly type sublanguage" probably should be "typed"
- page 4: "StongScript"
- page 5: "modelization" is neither an English word nor a technical term. I would suggest "modeling"
- page 8: In L4: "the object refereed to by x" should probably be "referred"
- page 9: "Expressions include..." this sentence should either drop the commas between the names of expression classes and the syntax examples, or should use semicolons to separate mention of different syntactic classes. As is, it's hard to parse.
- page 13: "case analyze is performed" instead of "analysis"
- page 13: "performans"
- page 15: "The definition of behcast(a,t,\sigma,K) are" (pluralization)
- page 16: "Wapper"
- page 17: First sentence of 5.6 has incorrect pluralization: "The monotonic and transient semantics both requires..."
- page 19: pluralization: "when d try to invoke method m" --> "tries"
- page 20: citations are technically not objects that a sentence can/should refer to ("described in [24]"), though we tend to do this in informal discussion. At a minimum, this makes it harder to follow discussion, since it's much easier to recall what Foobar et al. did than what [24] refers to.

Review #58C

Plain text

Overall merit

D. Reject

Reviewer expertise

Y. I am knowledgeable in this area, but not an expert

Paper summary

This paper presents a core class-based object calculus (called KafKa) as a means to model key aspects of gradual type systems. In particular, there are number of gradual type systems for objects that vary along many axes. The goal of this work is to expose these differences by defining translations to the core KafKa language. The result is several gradual type systems (i.e., TypeScript, Thorn, Transient Python, Typed Racket, and Monotonic Python) have been translated into KafKa for comparison.

Points For and Against

For

- It is important and interesting to study the design decisions of the various gradual type systems that have been proposed in order to elucidate the ramifications of those decisions.

- The core language is reasonable and the presentation clear for the most part.

Against

- The paper considers only a on-paper formalization of the translations. It provides no implementation as a means for gaining confidence in the translations.

Comments for author

The goal of the paper is interesting: to isolate the differences between various designs for gradual type systems as a means for motivating future investigation. To this end, the paper presents a core language KafKa that has the essential features necessary to compile different gradually-typed source languages. In particular, the differences most visibly manifest themselves as different casts in KafKa (structural, shallow, behavioral, and monotonic). Section 1 and Section 2 do a good job of giving intuition for these differences and given the necessary background to understand them. The table in Figure 1 provides a nice overview.

Section 3 provide translations of various gradual type systems by example, while Section 5 formalizes the translation with Section 4 providing a formalization of KafKa. All of these describes are clear and well described for the most part. This is nice.

My main concern with the paper as-is is that it does not provide a

result on which the community could gain confidence in these definitions. There are many possible ways to do this, but the paper needs to provide something along these lines before I could advocate for it.

One baseline validation would be an implementation of KafKa and compilers for the source languages into KafKa. This could be as minimal as a simple interprefer for KafKa--it need not be performant, along with compilers for reasonable subsets of the source language. This would enable at least conformance testing the translations by comparing executions of programs compiled to KafKa with those with their native execution environments.

Such an implementation could also enable providing a reference execution environment for comparing the performance of difference decision choices in gradually-typed language design. This could be an interesting way to isolate the performance impact of the language design from the differences in engineering present between the different execution environments.

Section 3.6 also hints at something potentially very interesting: designing a suite of litmus tests that could exhibit differences in gradually-typed language design.

Detailed Comments

- Page 10: The \sigma in the typing judgment form. It is not until 4.2 that we see \sigma as a heap.
- Page 10: broken reference
- Page 13: "performans"

- Page 14: effect -> affect
- Page 16: "Wapper"

Review #58D

Plain text

Overall merit

C. Weak reject

Reviewer expertise

Y. I am knowledgeable in this area, but not an expert

Paper summary

The paper presents a calculus that is intended to be used to compare the many gradual typing systems that have sprouted up in recent years. The system consists of a core calculus called Kafka. Kafka consists of a statically typed core augmented with an "any" type (`*`) and various dynamic operations:

- dynamic method dispatch `e@m(e)`:
  - looks up the method `m` dynamically
- two forms of structural cast:
  - `<t> a`, which checks that the object `a` is a subtype of `t`;
  - `<:t:> a` (chevrons), which checks that the object `a` has methods with suitable names, but doesn't check the types of those methods deeply.

Kafka's type system ensures that programs can only fail when performing one of these operations.

The paper defines a "gradual" superset of Kafka, designated by underlined expressions `_e_`. Each of the gradually typed languages (e.g., TypeScript, Thorn, Reticulated Python, etc) defines a set of rules to translate from the "gradual" superset to the more precise core Kafka calculus. These translations expose the difference in semantics between systems. For example, TypeScript translates the type of all variables to `*`, since it uses "optional" types, which means that a function `f` annotated with one type (say, `String`) may actually be supplied a value of another type (say, `Foo`), so long as `Foo` offers all "sufficiently compatible" versions of the methods that `f` will invoke. Stricter systems, such as Thorn, would translate a type declaration like `String` to a strict Kafka type (i.e., `String`).

Finally, to accommodate some of the more advanced "contractual" languages, notably Typed Racket and its derivatives, the gradual fragment of Kafka contains "behavioral" and "monotone" casts. During the translation step from gradual Kafka to core Kafka, these casts introduce dynamic class definitions that serve as proxies. These proxies monitor the types that cross between the statically and dynamically typed boundaries and detect errors.

Points For and Against

Pro:

- Defining a 'common vocabulary' for the various gradual typing systems makes it easier to compare their semantics.

Con:

- What is the paper's **key contribution**? The paper did not seem to present any new information or particular insights about existing systems, only perhaps to express the differences between systems in a particularly precise way.
- The calculus helped to clarify the dynamic semantics of the various gradual typing systems, but I found it less helpful in understanding their static semantics.
  - In other words, what set of programs are statically rejected? Are there interesting differences to highlight here as well?
- The "Litmus Tests" that distinguish between the various systems felt artificial; I would have hoped for tests that would

give a good feeling for the pros and cons of each approach.

Comments for author

One thing that the introduction of the paper makes clear is that the term "gradual typing" can be used to mean many different things. The promise of the paper then is that the Kafka framework can be used to effectively compare these distinct approaches, and I think that to some extent it succeeds.

However, I also had the feeling that the Kafka formalization itself seemed to add relatively little atop a more general comparison. Moreover, perhaps because the systems vary from one another, I had the impression that the features used by the various systems did not overlap so much with one another.

One question for clarification. Kafka offers only structural casts, but the JVM offers only nominal casts -- and I believe that some languages, such as Thorn, are also built on nominal casts. Naturally one can "model" nominal type systems from a structural one (e.g., by inserting a unique member), but I was surprised not to find any direct coverage of this in the section on Thorn.

Suggestions for clarifications

I found the distinction between "gradual Kafka" and "core Kafka" to be less clear than it could have been; I think partly this is because there is not one "gradual Kafka" but rather a family of them, as each "front end" introduces its own concepts (e.g., Thorn has "question" types like `?C`, and so forth). I would have appreciated a firmer separation between the two concepts. For example, on page 2, the paper states:

> Two different structural casts are built-in to Kafka...

then in the same paragraph it says:

> To support some of the more complex type systems, KafKa is extended by generative casts which create new wrapper classes.

I think what is happening here is that the "core Kafka" supports two forms of structural cast, but the "gradual Kafka" supports a richer set of casts (which are only used in Typed Racket).

Similarly, in the formal sections, I found the underlining sometimes confusing. For example, On page 13, rule THS3 includes a clause where the whole thing is underlined:

```
__m(t1) : D \in mtypes(C, K)__
```

Rule THS4 includes a very similar clause, but only some parts are underlined:

```
__m__(__t1__) : __?D__ \in mtypes(__C__, __K__)
```

I couldn't tell if this was meant to be conveying some sort of subtle distinction, or just a stylistic choice. (I sort of suspect that the formatting in THS3 is just a mistake, since other rules seem to prefer the THS4-style underlyining.)

Review #58E

Plain text

Overall merit	Reviewer expertise
D. Reject	X. I am an expert in this area

Paper summary

The paper describes KafKa, a class-based calculus with structural subtyping, both regular method dispatch and reflective dispatch, and several kinds of casts. The calculus is meant to be representative of the JVM and .NET CLR.

The paper describes translations to KafKa from calculi that model five gradually typed languages: TypeScript, Thorn, Typed Racket, Transient Python, and Monotonic Python. The purpose of these translations is to: 1) "capture the essence of gradual typing for objects" 2) "and highlight the challenges implementers face".

Points For and Against

+ The combination of class-based and structural subtyping is relatively unexplored in the literature, especially with the semantics of casts using in KafKa, so this is a point in favor of novelty.

- The paper is light on technical contributions. It does not present an evaluation of KafKa or of the translations. For example:

    - The paper does not give any meta theory for KafKa. Is KafKa type safe? Given the novel combination of casts, class-based method dispatching, and structural subtyping, there is a significant risk that it is not.
    - The calculi for the five languages being modeled do not seem to capture the important features of those five languages.
    - The paper presents five translations, but does not prove that the translations faithfully respect the operational semantics of the five languages being modeled. The paper does not prove that the translations are type preserving.
    - The paper does not present empirical results of any kind. It does not present performance results or case studies, etc.

- The second motivation for the translations is to highlight implementation challenges, so it is important for KafKa to faithfully represent the JVM and .NET CLR, but the paper does not seriously address this issue. Given that KafKa has structural subtyping, the difference seems non-trivial. For example, the paper could given a translation from KafKa to the JVM or the CRL.

- The first motivation for the translations is to "capture the essence" of five gradually typed languages in a common framework, but KafKa has to be extended with different kinds of casts to handle the different languages, and the translations for Typed Racket and Python are rather convoluted. In particular, the translation for Monotonic Python relies on generating proxies, but the point of monotonic was to remove the need for proxies.

- Regarding the faithfulness to the five languages, the paper presents them all as having bidirectional type systems, but many of the languages do not use bidirectional typing. Also, an important feature of Typed Racket and Python are their first-class classes, but that feature is not modeled here.

Comments for author

Abstract:

"cottage industry" => "illustrious research program"

"piecemeal" => "in a principled way"

1. Introduction

"Seik" => "Siek"

"The monotone cast <|t|> a returns a wrapper ..." Hmm, the point of monotonic references was to avoid wrappers.

2. Background

"ancestry is dominated by the work of Felleisen and his students" => "Felleisen and his students have made important contributions."

p. 4

"primitives are concretely typed they can be unboxed without tagging" Missing comma

p. 5

"transient semantics for Reticulated Python can be viewed as a worst case scenario for concrete types" Did you mean optional

types?

"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." Grammar problem, incomplete sentence

class A {m(x: C): A { this }n(x: *): C { x } } Hard to keep m and n straight, choose better names

3. Gradual Typing in Practice

p. 6

"So class A would translate to" But what about the rest of the program?

"casts are inserted on method entry and prior to returning" Transient does not insert casts prior to returning, but instead inserts a cast around the call to the method.

p. 8

"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."

Unclear whether this is faithful to monotonic, certainly not from a performance perspective .

p. 10

In the W7 rule, why is e' required to have type * instead of any type?

"The complete set of rules appear in Appendix section ??." Broken reference.

p. 11

"we have two judgments that perform cast insertion on expressions, following Pierce and Turner"

Why birectional typing?

p. 13

"The Transient semantics makes guarantee about what methods are avaible in the argument" Spelling and grammar

"... and return value of each typed method" No guarantees about the return value.

p. 15

"that is, being construction, of the correct type" Grammar problem.

"TRA2 and TRA3 are the most important cases of cast insertion;" But you only list TRA1 and TRA2 below.

p. 16

"one of the semantic Typed Racket" Grammar problem

p. 18

"whose method have is fully typed" grammar problem

"The purpose of the mon-MOS4 MOS2 Wrap function is further evidenced as the handling of calls to methods need to appropriately casted if the typing nature is unclear." grammar problem, missing "be"

"The monotonic cast <C>a imposes the type C onto the object at a and every object transitively reachable from a." This seems to say that every object reachable from object a has to have type C, which is probably not what is meant.

p. 19

"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."

This does not sound faithful to the monotonic semantics.

"The monotonic cast semantics in KafKa cannot be implemented using the in-place replacement technique"

So it seems that KafKa is a bad choice of intermediate language for implementing monotonic.

p. 20

"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"

That's bad!

Response

The author response should address reviewer questions and concerns, and correct misunderstandings. Make it short and to the point. Try to stay within 600 words. You may write more words but reviewers are only required to read the first 600 words, so address your key points first.

Markdown styling and LaTeX math supported

Cancel Preview Submit Save draft

600 words left