FOOL 2014

The Problem of Structural Type Tests in a Gradual-Typed Language

John Tang Boyland
University of Wisconsin-Milwaukee
boyland@uwm.edu

Abstract

The Grace programming language includes structural type tests and gradual typing. We demonstrate that this combination results in a mismatch. In particular, structural type tests (but not structural type assertions) can cause programs to give different results after typing annotations are added. We review the current uses of type tests in Grace programs and propose potential ways forward, all of which have their own drawbacks.

1. Introduction

Grace [2, 3] is a new language specifically tailored to teaching Computer Science. It has a simple syntax and a powerful yet minimal type system. For power and simplicity, it uses a structural type system. In order to make the system usable for rapid prototyping and scripting, Grace supports gradual typing.

Gradual typing [6] bridges dynamically and statically typed code by adding type assertions at the boundary between them. Even more important than implementation efficiency, static typing gives the ability to find errors sooner in the development process, either at compile-time, or dynamically, when execution enters a subsystem with particular explicit expectations. Furthermore, many gradually typed systems use types as contracts in which one can track *blame*, locating the code that breaks a contract [9].

In this paper, I argue that the combination of the three features (structural types, type tests and gradual types) leads to a serious problem. One cannot provide continuity between dynamically typed and statically typed programs, because adding a (valid) type annotation can change a program's behavior.

The problem I describe turns on how type tests are performed. In a nutshell, if more precise type information enables the type test to more precisely determine the type being tested, then a type test's result may switch from positive to negative (or vice versa). If a program can use the result of the type test in a conditional branch, then the addition of a type annotation can change its (non-error) behavior.

If, on the other hand, type tests only lead to possible runtime errors, and if type tests are implemented to err on the side of lenience in the case of missing type information, then the addition of a type annotation can only make a previously running program fail with a type error. It cannot change its non-error behavior.

I discuss this situation in the case of Grace, extending TinyGrace [4] (a statically-typed minimal dialect of Grace) with gradual features, keeping in mind the current (dynamically-typed) implementation, minigrace. Section 2 starts by presenting a definition of TinyGrace, modified to be more amenable to gradual typing. Then Section 3 discusses minigrace and considers an example of a type test in the example corpus of Grace programs distributed with minigrace. Section 4 discusses how to define a gradually-typed version of Grace, thus attempting to connect minigrace and TinyGrace. It gives a simple artificial program that demonstrates the problems with type tests. It also discusses the possibility of doing away with type tests altogether.

2. Syntax and Semantics

Figure 1 gives the syntax of TinyGrace, copied with minor variation from Jones and Noble [4] with an additional highlighted form for the "Dynamic" (or "Unknown") type. Jones and Noble further define a well-formedness relation on types that ensures that all recursive references are in scope and signatures have unique method names. In this paper, we assume these restrictions are fulfilled.

In examples as a shorthand and for clarity, I will use named type definitions and named objects in place of the recursive types and anonymous objects in Figure 1.

Figure 2 gives typing rules for TinyGrace, modified from Jones and Noble, by making the type system algorithmic (and dropping an explicit T-SUB rule) and in T-REQ, explicitly substituting the recursive type into the parameter types ($\overline{\tau_1}$) and the result type (τ_2).

For the current purposes, the type rule for matches, T-CASE, is the most relevant: the expression being matched over must be one of the types being matched (to make sure

$M ::= \texttt{method}\ S\ \texttt{\{}\ e\ \texttt{\}}$	*method*
$O ::= \texttt{object}\ \texttt{\{}\ \overline{M}\ \texttt{\}}$	*object creation*
$C ::= \texttt{case}\ \texttt{\{}\ x\ \texttt{:}\ \tau\ \texttt{->}\ e\ \texttt{\}}$	*branch*
$S ::= m\texttt{(}\overline{x\texttt{:}\tau}\texttt{)}\ \texttt{->}\ \tau$	*signature*
$e ::=$	*expression:*
O	*object*
x	*variable use*
$e\texttt{.}m\texttt{(}\overline{e}\texttt{)}$	*method call*
$\texttt{match}\ \texttt{(}e\texttt{)}\ \overline{C}$	*type test*
$\tau ::=$	*type:*
$\texttt{rec}\ X\texttt{.type}\ \texttt{\{}\ \overline{S}\ \texttt{\}}$	*object type*
X	*recursive reference*
$\tau\ \texttt{\|}\ \tau$	*union type*
$\tau\ \texttt{\&}\ \tau$	*intersection type*
$\texttt{?}$	*dynamic type*
$\Pi ::= \cdot \mid \Pi\texttt{,}\ X$	*type context*
$\Sigma ::= \cdot \mid \Sigma\texttt{,}\ \tau\ \texttt{<:}\ \tau$	*subtype context*
$\Gamma ::= \cdot \mid \Gamma\texttt{,}\ x\ \texttt{:}\ \tau$	*variable context*

Figure 1. TinyGrace syntax with an addition.

T-VAR
$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

T-OBJ
$$\frac{\tau = \texttt{rec X.type\{}\overline{m\texttt{(}\overline{x\texttt{:}\tau_1}\texttt{)->}\tau_2}\texttt{\}} \qquad \Gamma, \texttt{self} : \tau, \overline{x : \tau_1,} \vdash e : \tau_2}{\Gamma \vdash \texttt{object\{}\overline{\texttt{method}\ m\texttt{(}\overline{x\texttt{:}\tau_1}\texttt{)}\ \texttt{->}\ \tau_2\ \texttt{\{}\ e\ \texttt{\}}}\texttt{\}} : \tau}$$

T-REQ
$$\frac{\Gamma \vdash e_s : \texttt{rec}\ X\texttt{.type\{}\ \overline{S}\ \texttt{\}} \qquad m\texttt{(}\overline{x\texttt{:}\tau_1}\texttt{)->}\tau_2 \in \overline{S} \qquad \Gamma \vdash \overline{e_p : \tau_p} \qquad \overline{\tau_p \leq \tau_1[X \mapsto \texttt{rec}\ X\texttt{.type\{}\overline{S}\texttt{\}}]}}{\Gamma \vdash e_s\texttt{.}m\texttt{(}\overline{e_p}\texttt{)} : \tau_2[X \mapsto \texttt{rec}\ X\texttt{.type\{}\overline{S}\texttt{\}}]}$$

T-CASE
$$\frac{\Gamma \vdash e : \tau \qquad \tau \leq \mathrm{or}(\overline{\tau_1}) \qquad \overline{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}}{\Gamma \vdash \texttt{match(}e\texttt{)}\ \overline{\texttt{case}\ x\texttt{:}\tau_1\ \texttt{->}\ e_2} : \mathrm{or}(\overline{\tau_2})}$$

$$\mathrm{or}(\tau_1 \ldots \tau_n) = \tau_1 \,|\, \ldots \,|\, \tau_n$$

Figure 2. (Algorithmic) Type System for TinyGrace.

S-ASSUM
$$\frac{X \leq X' \in \Sigma}{\Sigma \vdash X \leq X'}$$

S-LIT
$$\frac{\Sigma, X \leq X' \vdash \overline{S \leq S'}}{\Sigma \vdash \texttt{rec}\ X\texttt{.type\{}\overline{S}\texttt{,}\overline{S_0}\texttt{\}} \leq \texttt{rec}\ X'\texttt{.type\{}\overline{S'}\texttt{\}}}$$

S-UNIONL
$$\frac{\Sigma \vdash \tau \leq \tau_1}{\Sigma \vdash \tau \leq \tau_1 | \tau_2}$$

S-UNIONR
$$\frac{\Sigma \vdash \tau \leq \tau_2}{\Sigma \vdash \tau \leq \tau_1 | \tau_2}$$

S-UNION
$$\frac{\Sigma \vdash \tau_1 \leq \tau \qquad \Sigma \vdash \tau_2 \leq \tau}{\Sigma \vdash \tau_1 | \tau_2 \leq \tau}$$

S-INTERL
$$\frac{\Sigma \vdash \tau_1 \leq \tau}{\Sigma \vdash \tau_1 \& \tau_2 \leq \tau}$$

S-INTERR
$$\frac{\Sigma \vdash \tau_2 \leq \tau}{\Sigma \vdash \tau_1 \& \tau_2 \leq \tau}$$

S-INTER
$$\frac{\Sigma \vdash \tau \leq \tau_1 \qquad \Sigma \vdash \tau \leq \tau_2}{\Sigma \vdash \tau \leq \tau_1 \& \tau_2}$$

S-SIG
$$\frac{\Sigma \vdash \overline{\tau_1' \leq \tau_1} \qquad \Sigma \vdash \tau_2 \leq \tau_2'}{\Sigma \vdash m\texttt{(}\overline{x\texttt{:}\tau_1}\texttt{)->}\tau_2 \leq m\texttt{(}\overline{x'\texttt{:}\tau_1'}\texttt{)->}\tau_2'}$$

Figure 3. Subtyping in TinyGrace: $\Sigma \vdash \tau \leq \tau$.

one of the branches match), and in the body of the case, one can assume that the match variable has the chosen type. The resulting type is the union of all the body types.

The subtyping relation $\tau \leq \tau$ is defined in Fig. 3. It differs significantly from that of Jones and Noble only in the (stricter) way recursive types are handled. Here we use the "Amber" rule for simplicity, but since recursive types aren't the focus of this work, the difference is irrelevant. Following Jones and Noble, in S-LIT, the signatures are assumed to be freely reorderable so that we can assume that all the matching methods are in the same order at the front of the type. For now, since TinyGrace doesn't use the dynamic type ?, we define no subtyping rules for ?.

Jones and Noble concede the subtyping of intersection types is not complete because a signature with a single method whose return type is an intersection type is not a supertype of the intersection of two object types with signatures with the same method having the individual return types. The same argument applies *mutatis mutandi* to union types, although Jones and Noble do not mention this fact.

Figure 4 repeats evaluation rules from Jones and Noble, with the addition of "type assertions" on actual parameters in R-REQ to match the semantics implemented by minigrace.[1] The *type assertion* $O \in \tau$ consists of getting the type of the object and then testing the resulting type with subtyping. The static type system of TinyGrace ensures that the type

[1] It would also make sense to add type assertions on the method return, but as this would require more machinery (an explicit cast syntax) and because minigrace currently doesn't check return values, I do not do so in this paper.

R-RECV

$$\frac{e_1 \longrightarrow e_1'}{e_1\texttt{.}m\texttt{(}\overline{e_2}\texttt{)} \longrightarrow e_1'\texttt{.}m\texttt{(}\overline{e_2}\texttt{)}}$$

R-PRM

$$\frac{e_2 \longrightarrow e_2'}{e_1\texttt{.}m\texttt{(}\overline{O}\texttt{,}e_2\texttt{,}\overline{e_3}\texttt{)} \longrightarrow e_1\texttt{.}m\texttt{(}\overline{O}\texttt{,}e_2'\texttt{,}\overline{e_3}\texttt{)}}$$

R-REQ

$$\frac{\texttt{(method }m\texttt{(}\overline{x\texttt{:}\tau_1}\texttt{) -> }\tau_2\texttt{ \{}e\texttt{\})} \in M \qquad \overline{O \in \tau_1}}{\texttt{object\{}\overline{M}\texttt{\}.}m\texttt{(}\overline{O}\texttt{)} \longrightarrow e[\texttt{self} \mapsto \texttt{object\{}\overline{M}\texttt{\}}, \overline{x \mapsto O}]}$$

R-MATCH

$$\frac{e \longrightarrow e'}{\texttt{match (}e\texttt{) }\overline{C} \longrightarrow \texttt{match (}e'\texttt{) }\overline{C}}$$

R-CASE

$$\frac{O \in \tau}{\texttt{match (}O\texttt{) case \{}x\texttt{:}\tau\texttt{->}e\texttt{\}}\overline{C} \longrightarrow e[x \mapsto O]}$$

R-MISS

$$\frac{O \notin \tau}{\texttt{match (}O\texttt{) case \{}x\texttt{:}\tau\texttt{->}e\texttt{\}}\overline{C} \longrightarrow \texttt{match (}O\texttt{) case }\overline{C}}$$

Figure 4. Evaluation for TinyGrace (and minigrace).

R-INST

$$\frac{\texttt{rec }X\texttt{.type\{}\overline{S}\texttt{\}} \le \tau}{\texttt{object\{}\overline{\texttt{method }S\texttt{\{}e\texttt{\}}}\texttt{\}} \in \tau}$$

Figure 5. Evaluation of Type Assertions for TinyGrace.

assertions that I added to R-REQ always succeed, and that a match always has a case that works.

The rule R-MISS uses the negation of of a type assertion to determine whether to move on to the next case. Not surprisingly, this negation lies at the heart of the problem of combining case matching with gradual typing.

3. Dynamically-Typed Grace

Grace is being used in instruction for the first time in Fall 2014[2] using its dynamically-typed implementation, minigrace. The implementation supports a language that is a superset of TinyGrace with many additional features (including primitives, mutable state and much more). In particular, types are not checked statically, and in any case are optional: type annotations of the form "`:` τ" or "`->` τ" can be omitted. In this paper, however, I will use "?" as the dynamic type, and so share the syntax figure (Fig. 1)

[2] At Pomona College with Kim Bruce.

$$\frac{L \supseteq \{\overline{m}\}}{L \le \texttt{rec }X\texttt{.type\{}\overline{m\texttt{(}\overline{\tau_1}\texttt{)->}\tau_2}\texttt{\}}} \qquad \frac{L \le \tau_1}{L \le \tau_1 | \tau_2}$$

$$\frac{L \le \tau_2}{L \le \tau_1 | \tau_2} \qquad \frac{L \le \tau_1 \qquad L \le \tau_2}{L \le \tau_1 \& \tau_2}$$

R-INST'

$$\frac{\{\overline{m}\} \le \tau}{\texttt{object\{}\overline{\texttt{method }m\texttt{(}\ldots\texttt{)->}\ldots\texttt{\{}e\texttt{\}}}\texttt{\}} \in \tau}$$

Figure 6. Evaluation of Type Assertions for minigrace.

Being dynamically-typed, minigrace does not use a static type system (no equivalent of Fig. 2), but (in my featherweight conception of minigrace), it shares the evaluation relation given in Fig. 4, with one difference seen in Fig. 6. Assertions are implemented by simply checking whether the set of labels in the object covers those required by the type.[3] An important property of this system is that the evaluation of type assertions do not depend on type annotations on methods of the object being tested or on methods of the type being test against.

As a result, one property enjoyed by minigrace is that type annotations cannot change the no-error semantics of a program. As just explained, adding a type annotation makes no difference in the evaluation of "`match`" expressions, and can only lead to evaluation getting stuck in R-REQ. This property is highly desirable in a gradually typed language, although minigrace is not a true gradually typed language implementation, but rather dynamic typing with the addition of type assertions. Vitousek and others [8] observe that simply adding dynamic type assertions to a dynamically typed language is not sufficient to detect all contract failures. And without static type checking, one cannot achieve even the relative type-safety of a gradually-typed language.

Almost all Grace programs that are currently being used are untyped since minigrace does not perform any static typing. For instance, in the entire corpus (3000 lines) of demonstration programs for Kim Bruce's programming class at Pomona, there is not a single use of `match` that tests a type (the few uses of `match` are used for value matching). This makes it harder to find motivating examples.

The syntax of Grace's catch clauses hints that (as with Java), type tests are used, but a fuller reading of the current draft specifiecation [3] indicates that exceptions are classified by "kinds" which are simply objects created in a controlled way from a base kind. In other words, there are no user-defined exception classes.

Type tests are actually used for ad hoc polymorphism, for example (in code from the minigrace compiler):

[3] Personal communication, Michael Homer.

```
type LinePos = {
    line -> Number
    linePos -> Number
}
type RangeSuggestions = {
    line -> Number
    posStart -> Number
    posEnd -> Number
    suggestions
}
...
match (e.data)
  case { lp : LinePos -> ...
      atPosition(e.data.line, e.data.linePos)
  }
  case { rs : RangeSuggestions -> ...
      atRange(rs.line, rs.posStart, rs.posEnd)
      ...
  }
  case { _ -> }
```

The behavior of this code depends on whether the data in the object e is a line position or a range suggestion. The empty clause at the end prevents an error from being raised, at the cost of silently doing nothing.

4. Gradual Typing

In this section, I describe some ways to define gradual typing for Grace. First, I describe how TinyGrace can be made gradual and then how minigrace can incorporate static typing. For each proposal, I show examples that demonstrate a problem with the proposal.

The issues discussed here are not an issue for a nominal type system (such as with Java) because testing membership in a class simply uses the class named at creation (and its declared supertypes) and does not need to consider the methods of the instance (let alone their declared types).

4.1 Adding dynamic types to TinyGrace

The static type system currently does not permit anything of declared dynamic type (?) to be used. This can be changed by updating subtyping by adding two rules:

S-DynTop
$$\frac{}{\Sigma \vdash \tau \leq ?}$$

S-DynBot
$$\frac{}{\Sigma \vdash ? \leq \tau}$$

These rules add cycles to the subtyping relation, and cause it not to be transitive, but these defects can be overlooked since the type system is algorithmic, lacking a T-Sub rule.

Siek and Taha [6] take a different approach. Instead of changing subtyping, they define a new relation $\lesssim$ that combines subtyping with consistency "$\sim$."[4] This relation is then used with T-Req (and in type assertions) in the place of normal subtyping, with much the same effect.

[4] I conjecture that $\lesssim$ is the same as $\leq$ with the addition of the new rules added in this section.

```
type Object = { }
type Boolean = {
  choose(o1 : Object, o2 : Object) -> Object
}
type Nat = {
  isZero() -> Boolean
  pred() -> Nat
  succ() -> Nat
}
type Window =  {
  draw(b : Boolean) -> Object
}
type Cowboy = {
  draw(n : Nat) -> Cowboy
}

object a = {  method a() -> Object { self } }
object b = {  method b() -> Object { self } }
object c = {  method c() -> Object { self } }

match (object {
         method draw(x : Nat ) -> Cowboy {
           self
         }
       })
  case { w:Window -> a }
  case { c:Cowboy -> b }
  case { o:Object -> c }
```

Figure 7. Problematic Code using Type Tests

These additions make TinyGrace's type assertions more lenient if a type annotation is removed, that is, more likely to succeed. This is a problem for R-Miss because removing a type annotation can cause this rule to become *less* applicable. For example, if the code in Fig. 7 is executed in TinyGrace, the result is "b." But if the highlighted type annotations (on the match subject) are replaced with "?" then the the match subject could be seen as either a Window or a Cowboy, and since the first case is checked first, the result will be "a." Thus we see that the no-error behavior of this program depends on the presence of type annotations.

Vitousek and others [8] define three ways of adding gradual types, distinguished mainly by how they handle type assertions. At the danger of over-simplification, these three techniques are as follows:

guarded Objects with dynamic type passed to methods are wrapped to check arguments and results of its uses.

$$|?| = |?|$$
$$|\tau_1 \,|\, \tau_2| = |\tau_1| \ \texttt{|} \ |\tau_2|$$
$$|\tau_1 \& \tau_2| = |\tau_1| \ \texttt{\&} \ |\tau_2|$$
$$|\texttt{rec}X\texttt{.type\{}\overline{m(\overline{x\texttt{:}\tau})\texttt{->}\tau'}\texttt{\}}| = \texttt{rec}X\texttt{.type\{}\overline{m(\overline{x\texttt{:?}})\texttt{->?}}\texttt{\}}$$

Figure 8. Type Erasure: $|\tau| = \tau'$.

transient Every use of an object is checked against its declared type.

monotonic Every type assertion on an object is put into effect for all further uses of the object.

The example in Fig. 7 has the same behavior with all three systems, mainly because the object of dynamic type is used only in `match`. Of course a full definition of gradual typing derived from TinyGrace, using these techniques would need to handle their semantics: the "guarded" approach would require the creation and handling of proxies; the monotonic approach would see a type assertion as a side-effect on the state of the object, thus requiring tracking of mutable object state.

Another example of the problem of having semantics depend on type annotations is type inference. If a tool is used to automatically infer (say) return types for methods, this may cause a program to execute differently. In a system only using type assertions in a positive way, this problem does not occur.

4.2 Extending minigrace into a gradual system

Next, we turn to extending minigrace into a true gradual implementation. The main difficulty adding the static type rules on top of minigrace is that type assertions merely test the presence of certain labels and do not ensure even that the methods have the correct arity, and the fact that types are ignored means that we will need to use TinyGrace's type assertion alongside that of minigrace to do the checks needed for gradual typing.

Thus I suggest that a updated version of *minigrace* use TinyGrace's type system, evaluation system and assertion semantics (and the two new subtyping rules above), with a simple change to the premise of the R-MISS rule:

R-MISS'

$$\frac{O \notin |\tau|}{\texttt{match (}O\texttt{) case \{}x\texttt{:}\tau\texttt{->}e\texttt{\}}\overline{C} \longrightarrow \texttt{match (}O\texttt{) case } \overline{C}}$$

Here $|\tau|$ is the "type erasure" of τ (see Fig. 8).

Recall that the negative use of a type assertion is the cause of the problem with type annotations causing changes in the no-error behavior of programs. By using the erased type in R-MISS, we maintain consistency. In the case of the problematic code in Fig. 7, the evaluation without the annotations on the match subject is "a" but the evaluation of the fully-typed program is an error since the second branch can only be chosen if the subject cannot match the erased Window type (which is identical to the erased Cowboy type).

```
type Object = { }
type Number = { ... }
type Point = {
  x -> Number
  y -> Number
}
type Vector = {
  dx -> Number
  dy -> Number
}

...
method magn(o : Object) -> Number {
  match(o)
    case { p : Point ->
           sqrt(p.x * p.x + p.y * p.y) }
    case { v : Vector ->
           sqrt(v.dx * v.dx + v.dy * v.dy) }
    case { _ : Object -> 0 }
}
```

Figure 9. Type tests used for ad hoc polymorphism.

While this gradual system does ensure that the no-error semantics of programs is unaffected by type annotation, it does suffer from the fact that a well-typed program (such as Fig. 7 or Fig. 9) can get stuck at run-time. For an example for Fig. 9, consider calling the `magn` method with an object that has `x` and `y` fields of type String. The first branch would not match, but the modified R-MISS' rule would not allow other branches to be chosen.

So, this updated version of minigrace has most of type safety, including type preservation, but lacks progress for type matches, which can now fail to find a match in the presence of ambiguity in erased selection. Full type safety can only be retained using type negation which would give safe but complex types to `match` expressions.

4.3 Removal

A third, more drastic proposal would be to remove type tests from Grace altogether. The `match` syntax would remain but only for values and patterns. Gradual typing could be used to get the effect of a checked cast: `cast(...)` could be wrapped around any expression whose type doesn't match its context, where `cast` has the following definition:

```
method cast(x : ?) -> ? { x }
```

Grace will have generics, and assuming that the dynamic semantics includes assertions on return values as well, one could instead define `cast` generically:

```
method cast<T>(x : ?) -> T { x }
```

Of course, without type tests, there is no equivalent of "instanceof" or else we would again have the situation where no-error semantics could depend on type annotations. And thus the language without type tests is strictly less powerful.

Are type tests actually useful? Certainly Java programs make extensive use of instanceof, which can be seen in the equals methods of even very simple classes. Grace however provides equality as a built-in, and supporting ad hoc polymorphism (static or dynamic overloading) is a weak reason to include a feature. Usually defining a common interface is cleaner than selecting a different behavior depending on the type of a value.

Furthermore, type tests interfere with parametricity. In Reticulated Python, type tests allow users to see supposedly hidden proxy objects created in the Guarded implementation [8]. If Grace were to abandon type tests, one might be able to build strong guarantees about abstraction. Indeed, type tests can be seen as a weak form of reflection, breaking abstraction in a useful way.[5]

4.4 Summary

There seem to be two ways to define gradual typing in Grace: one to extend TinyGrace with dynamic types, in which case type annotations can change the program's semantics, and one in which minigrace is modified to check types more precisely, but where statically typed programs can still get stuck.

The problem is due to a negative use of a type assertion in legal evaluation (R-MISS). If the type assertion depends on type annotations (TinyGrace), program semantics are changed by annotations. If it does not depend on type annotations, it cannot guarantee type safety (as needed by T-CASE). I am currently tempted to propose eliminating the problematic construct altogether.

5. Related Work

I claim that gradual typing has as a goal the preservation of semantics even as type annotations are added. For various reasons, I have not seen this goal explicitly identified in related work. Siek and Taha [6] come close, listing as the first (albeit not necessarily primary) goal of gradual typing:

> Programmers may omit type annotations on parameters and immediately run the program; run-time type checks are performed to preserve type safety.

Arguably "may omit type annotations" implies that the programs behavior should not change if this is done.

A formal statement of the property I claim should be a goal of gradual typing relies on a definition of "partial erasure" of annotations: I write $e \prec e'$ if e' is the same as e except that 0 or more types are replaced with "?."

[5] Andrew Black, personal communication

PROPERTY 5.1 (Gradual Guarantee). *If an expression e_1 evaluates without error one step to e_2, then any expression e'_1 with fewer annotations ($e_1 \prec e'_1$) also evaluates in zero or more steps to e'_2 where $e_2 \prec e'_2$.*

The property of course would need to be modified for other languages to handle mutable state and other aspects, but hopefully the spirit of the concept is clear. In this section, we review other gradual languages to see whether they provide this guarantee.

Siek and Taha's original gradual type system [5] (for functional languages) as well as their OO system [6] apparently both provide the gradual guarantee. Applying these ideas to a real language in Python [8], Siek (with others) investigated the three ways mentioned above to implement the dynamic checks. The guarded approach used proxies and suffered in two ways: the proxies were visible both at the object identity level as well as at the type level (as mentioned previously). As a result, the guarded system did *not* provide the gradual guarantee. The authors clearly saw this lack as a problem, which supports my thesis.

TypeScript [1] does not achieve type soundness, but otherwise resembles a gradually-typed programming language. It provides extra features not provided by JavaScript (the untyped variant) but compiles down to (unannotated) JavaScript. It uses "full erasure" in that the types cannot be detected at run-time. In particular, type casts are *not* implemented by dynamic checks. Thus it appears that TypeScript supports the gradual guarantee.

Swamy and others [7] define a full gradually-typed version of JavaScript. They include reflection functions reminiscent of type tests: isTag<t> and canTag<t>. The former function permits the program to query the current (monotonic) type of an object and the latter checks to see if it could be legally updated. These functions indeed violate my gradual guarantee.

Java, while not a gradual system, provides type annotations that are also accessible through reflection. If Grace's type tests are seen as a reflection API, then the fact that they do not satisfy the gradual guarantee is less disturbing.

6. Conclusions

Structural type tests have a surprising interaction with gradual typing which makes the combination somewhat distasteful. The cleanest solution seems to me to remove type tests from the language, at the cost of expressive power.

While writing this paper, I was at first enamored of regularizing the minigrace approach, but the way in which this can cause type errors in fully statically-typed code makes this option less attractive to me.

Continuing with the semantics of type tests from TinyGrace is also unattractive, especially when one considers how type inference can exacerbate the problem, by changing the semantics of a program.

In conclusion, I see no obvious solution to the problem and can only hope that type tests will not be widely used.

Acknowledgments

I thank Andrew Black, Kim Bruce and James Noble for discussing the issue with me and encouraging me to write up my vague worries. I thank Michael Homer and Tim Jones for helping me figure out how to execute minigrace, and for clarifications on the intended semantics of minigrace. I thank the anonymous reviewers for insightful comments that led to major changes in the paper.

References

[1] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding TypeScript. In Richard Jones, editor, *ECOOP'14 — Object-Oriented Programming, 28th European Conference*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer, Berlin, Heidelberg, New York, 2014.

[2] Andrew P. Black, Kim B. Bruce, Michael Homer, and James Noble. Grace: The absence of (inessential) difficulty. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! '12, pages 85–98. ACM, New York, NY, USA, 2012.

[3] Andrew P. Black, Kim B. Bruce, and James Noble. The Grace programming language, draft specification version 0.5.1853. August 2014.

[4] Timothy Jones and James Noble. Tinygrace: A simple, safe, and structurally typed language. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14, pages 3:1–3:6. ACM, New York, NY, USA, 2014.

[5] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming*, pages 81–92. September 2006.

[6] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *ECOOP'07 — Object-Oriented Programming, 21st European Conference*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer, Berlin, Heidelberg, New York, 2007.

[7] Nikhil Swamy, Cedric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin Bierman. Gradual typing embedded securely in javascript. In *Conference Record of POPL 2014: the 41st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 425–438. 2014.

[8] Michael Vitousek, Andrew Kent, Jeremy Siek, and Jim Baker. Abstracting design and evaluation of gradual typing for Python. Presented at Dynamic Language Symposium, 2014.

[9] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In Giuseppe Castagna, editor, *ESOP'09 — Programming Languages and Systems, 18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer, Berlin, Heidelberg, New York, 2009.