Gradual Types as Error Suppression

A Constructive View of Type Warnings

LILY BROWN, ANDY FRIESEN, and ALAN JEFFREY, Roblox, USA

This paper presents the view of gradual typing adopted by the Luau programming language. Prior work on gradual typing has been based on *type compatibility*, that is a relation on types $T \sim U$ given by contextually closing $T \sim \mathsf{any} \sim U$. Type systems based on type compatibility use $\sim$ rather than type equivalence (or a similar presentation for languages with subtyping). We take a different tack, which is to view type warnings *constructively* as a proof object $\mathrm{Warn}(\Gamma \vdash M : T)$ saying that the type derivation $\Gamma \vdash M : T$ should generate a warning. Viewing type warnings constructively allows us to talk about *error suppression*, for example the any type is error suppressing, and so this type system is gradual in the sense that developers can explicitly annotate terms with the any type to switch off type warnings. This system has the usual "well-typed programs don't go wrong" result for program which do not have explicit type annotations with error suppressing types, except this property can now be stated as the presence of $\mathrm{Warn}(\Gamma \vdash M : T)$ rather than the absence of a run-time error. This system has been deployed as part of the Luau programming language, used by millions of users of Roblox Studio.

CCS Concepts: • **Software and its engineering** → **Semantics**.

ACM Reference Format:
Lily Brown, Andy Friesen, and Alan Jeffrey. 2025. Gradual Types as Error Suppression: A Constructive View of Type Warnings. *Proc. ACM Program. Lang.* 9, POPL, Article ?? (January 2025), 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Gradual Typing

The aim of *gradual typing* [3, 4] is to allow a code base to migrate from being untyped to being typed. This is achieved by introducing a type any (also called ? or $*$) which is used as the type of a expression which is not subject to type checking. For example, in Luau, a variable can be declared as having type any, which is not subject to type checking:

```
local x : any = "hi"
print(math.abs(x))
```

This program generates a run-time error, but because x is declared has having type any, no type error is generated. Similarly, any expression can be cast to having type any, which is not subject to type checking:

```
print(math.abs("hi" :: any))
```

Again, this program generates a run-time error, but because "hi" is cast to having type any, no type error is generated.

Prior work on gradual typing has been based on *type compatibility*, that is a relation on types $T \sim U$ given by contextually closing $T \sim \mathsf{any} \sim U$. Type systems based on type compatibility use $\sim$ rather than type equivalence (or a similar presentation for languages with subtyping).

Authors' address: Lily Brown; Andy Friesen; Alan Jeffrey, Roblox, San Mateo, CA, USA.

2475-1421/2025/1-ART?? $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

For example, the type rules for function application in [3] are:

$$\frac{\Gamma \vdash M : \mathsf{any} \quad \Gamma \vdash N : U}{\Gamma \vdash M(N) : \mathsf{any}}\,(\textsc{GApp1}) \qquad \frac{\Gamma \vdash M : (S \to T) \quad \Gamma \vdash N : U \quad S \sim U}{\Gamma \vdash M(N) : T}\,(\textsc{GApp2})$$

which requires that the argument type is *compatible* with (rather than equal to) to the function source type. For example:

$$\frac{\Gamma \vdash \mathtt{math.abs} : (\mathtt{number} \to \mathtt{number}) \quad \Gamma \vdash x : \mathsf{any} \quad \mathtt{number} \sim \mathsf{any}}{\Gamma \vdash \mathtt{math.abs}(x) : \mathtt{number}}$$

The problem we discovered in implementing gradual typing on top of type compatibility is that it is a source of subtle bug, because the type system is very sensitive as to when type equality is used rather than type compatibility. For example comparing two type rules:

$$\frac{\Gamma \vdash M : F \quad \Gamma \vdash N : U \quad F = (S \to T) \quad S \sim U}{\Gamma \vdash M(N) : T}\,(\textsc{GApp2}') \qquad \frac{\Gamma \vdash M : F \quad \Gamma \vdash N : U \quad F \sim (S \to T) \quad S \sim U}{\Gamma \vdash M(N) : T}\,(\textsc{GApp2}'')$$

These rules only differ in whether they use type compatibility rater than equality, but they have very different semantics. Rule GApp2′ is the same as GApp2′ (and so is sound) but using GApp2″ we can derive:

$$\frac{\Gamma \vdash \mathtt{math.abs} : (\mathtt{number} \to \mathtt{number}) \quad \Gamma \vdash x : \mathtt{string} \quad (\mathtt{number} \to \mathtt{number}) \sim (\mathsf{any} \to \mathtt{number}) \quad \mathsf{any} \sim \mathtt{string}}{\Gamma \vdash \mathtt{math.abs}(x) : \mathtt{number}}$$

which is unsound. Problems like this come down eventually to the fact that $\sim$ is not transitive, and in the presence of other features such as unification and subtyping gave rise to subtle bugs (for example code that assumed that unification solved for type equality rather than compatibility).

1.2 Error Suppressing Types

There as been a long history of improving type errors reported to users, going back to the 1980s [1, 6]. One source of pain in type error reporting is *cascading* type errors, for example:

```
local x = "hi"
local y = math.abs(x)
local z = string.lower(y)
```

In this case `math.abs(x)` should generate a type error, since the type `string` is inferred for `x`, and the type of `math.abs` is `number` → `number`. It is not obvious whether a type error should be generated for `string.lower(y)`. If the type `number` is inferred for `y`, then an error should be reported, since the type of `string.lower` is `string` → `string`. But this will not be the best user experience, since this will give a common experience of multiple cascading type errors, of which only the first error is genuine.

One heuristic to eliminate cascading errors is to mark the type of any expression which causes a type error to be emitted as *error suppressing*. Error suppressing types are then used to percolate the information that a type error has already been generated, and so avoid cascading type errors.

For example, in Luau, a type `error` is introduced, and any type T which is a supertype of `error` is considered to be error suppressing. For instance, the types inferred for the above program are:

```
local x : string = "hi"
local y : number | error = math.abs(x)
local z : string | error = string.lower(y)
```

Since `number` | `error` is an error suppressing type, `string.lower(y)` will not report a type error.

The existence of `error` explains why Luau, in common with TypeScript [2], has both an error suppressing type `any` and a non-error suppressing type `unknown`. `any` is the top type, and `unknown` is the top non-error-suppressing type. We consider `any` to be equivalent to `unknown` | `error`.

Error suppressing types as a technique for minimizing cascading type errors appears to folklore, for example it is implemented in Typed Racket [5], but does not appear to have been academically published.

1.3 Gradual Typing via Error Suppressing Types

1.4 Constructive Type Errors

1.5 Contributions

2 FURTHER WORK

TODO

A PRAGMATIC SEMANTIC SUBTYPING

TODO

REFERENCES

[1] G. F. Johnson and J. A. Walz. 1986. A Maximum Flow Approach to Anomaly Isolation. 44–57. https://doi.org/10.1145/512644.512649

[2] Microsoft. 2023. TypeScript. https://www.typescriptlang.org/

[3] J. G. Siek and W. Taha. 2006. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*. 81–92.

[4] J. G. Siek and W. Taha. 2007. Gradual Typing for Objects. In *Proc. European Conf Object-Oriented Programming*. 2–27.

[5] S. Tobin-Hochstadt. 2008. New Error Handling for Type Parsing Errors. https://github.com/racket/racket/commit/3a9928474523b042f83a7a707346daa01ef63899 Commit to Typed Racket.

[6] M. Wand. 1986. Finding the Source of Type Errors. In *Proc. Principles of Programming Languages*. 38–43. https://doi.org/10.1145/512644.512648