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), 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The aim of *gradual typing* [1, 2] 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 [1] are:

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

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

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

2 FURTHER WORK

TODO

A PRAGMATIC SEMANTIC SUBTYPING

TODO

REFERENCES

[1] J. G. Siek and W. Taha. 2006. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*. 81–92.
[2] J. G. Siek and W. Taha. 2007. Gradual Typing for Objects. In *Proc. European Conf Object-Oriented Programming*. 2–27.