# 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), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

### 1.1 Gradual Typing

The aim of *gradual typing* [5, 6] 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 [5] 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:

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

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:

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

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 [2, 8]. 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.

Error suppressing types as a technique for minimizing cascading type errors appears to folklore, for example it is implemented in Typed Racket [7], but does not appear to have been academically published.

### 1.3 Gradual Typing via Error Suppressing Types

At this point a reader might guess where this paper is going. Gradual types are well established research area which allows programmers to type expressions with type `any` to suppress type errors. Error suppressing types are well established folklore which allows type inference to type expressions with error suppressing types to suppress cascading type errors.

The connection between these two areas is pretty obvious, but has not been made before. If `any` is an error suppressing type, then gradual typing is very similar to error suppression, but does not require the non-transitive type compatibility relation.

In Luau, the use of error suppressing types explains why Luau, in common with TypeScript [4], 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`.

### 1.4 Constructive Type Errors

Error suppressing types is a folklore implementation technique for suppressing cascading type errors. In this paper we present a formalization as a constructive model of type errors. We do this by separating out type derivations fro type errors.

For example, one traditional way to present the type rule for function application is:

$$\frac{\begin{array}{l}\Gamma \vdash M : T \\ \Gamma \vdash N : U \\ U <: \mathsf{src}(T)\end{array}}{\Gamma \vdash M(N) : \mathsf{apply}(T, U)}$$

using functions to calculate the domain of a function $\mathsf{src}(T)$, for example following [3, §5.2]:

$$\mathsf{src}(S \to T) = S \quad \mathsf{src}(S \cap T) = \mathsf{src}(S) \cup \mathsf{src}(T) \quad \cdots$$

and to calculate the result type of applying a function $\mathsf{apply}(T, U)$, for example following [3, §5.3]:

$$\mathsf{apply}(S \to T, U) = \begin{cases} T & \text{if } U <: T \\ \mathsf{any} & \text{otherwise} \end{cases} \qquad \mathsf{apply}(S \cap T, U) = \mathsf{apply}(S, U) \cap \mathsf{apply}(T, U) \quad \cdots$$

(The details of these function are spelled out in Appendix A.)

This formulation is fine if is one is only interested in well typed programs, and in not in the behavior of badly typed programs. For example, the classic "well typed programs don't go wrong" can be stated as:

$$(M \to^* M') \to (\mathsf{RunTimeErr}(M')) \to \neg(\emptyset \vdash M : T)$$

(for an appropriate definition of $\mathsf{RunTimeErr}(M')$). Now, this does not have a constructive model of type warnings, since we are only considering badly typed programs as ones where $\neg(\emptyset \vdash M : T)$.

In this paper, we separate type derivations from their type errors. This is done by explicitly tracking the derivation tree for typing, for example:

$$\frac{\begin{array}{c}D_1 : (\Gamma \vdash M : T) \\ D_2 : (\Gamma \vdash N : U)\end{array}}{\mathsf{app}(D_1, D_2) : (\Gamma \vdash M(N) : \mathsf{apply}(T, U))}$$

which allows us to define which type derivations generate warnings, for example there are three ways a warning can be generated from a function application $M(N)$, bubbling up a warning from $M$ or $N$, or by a failure of subtyping:

$$\frac{\mathsf{Warn}(D_1)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))} \qquad \frac{\mathsf{Warn}(D_2)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))} \qquad \frac{U \not<: \mathsf{src}(T)}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))}$$

This is based on a constructive framing of failure of subtyping, fleshed out in Appendix A:

$$\frac{\begin{array}{c}v \in [\![T]\!] \\ v \in [\![U]\!]^{\complement}\end{array}}{T \not<: U}$$

Now, there are two important results about this presentation of constructive type errors. The first is *infallible typing*, that every program can be type checked in every context:

$$\forall \Gamma, M \,.\, \exists T \,.\, (\Gamma \vdash M : T)$$

(We write $\mathsf{typeof}(\Gamma, M)$ for this derivation tree.) The second is that we can state "well typed programs don't go wrong" constructively:

$$(M \longrightarrow^* M') \longrightarrow (\mathsf{RunTimeErr}(M')) \longrightarrow \mathsf{Warn}(\mathsf{typeof}(\emptyset, M))$$

Since this is phased constructively, Curry Howard means we can think of this statement in two ways:

- as a proof that "well typed programs don't go wrong"
- as a time-travel debugger, that for any execution $(M \longrightarrow^* M')$ where $M'$ has a run-time error, it can be run back in time to find a root cause type error for $M'$.

Constructive type errors can be easily adapted to error suppressing types, for example we only report a failure of subtyping when both the type of the function and the type of argument are not error suppressing:

$$\frac{\begin{array}{c}\mathtt{error} \not<: T \\ \mathtt{error} \not<: U \\ U \not<: \mathsf{src}(T)\end{array}}{\mathsf{Warn}(\mathsf{app}(D_1, D_2))}$$

Now, in general this breaks type soundness, because if every type is any then all type errors are suppressed. But we can show that in the case of a program in which no types are error suppressing, "well typed programs don't go wrong".

### 1.5 Contributions

In summary, this paper combines two well-explored areas:

- gradual types, and
- error suppressing types.

The new contributions of this paper are:

- combining gradual types and error suppressing types,
- formalizing error suppressing types as constructive type errors, and
- showing type soundness for programs which do not contain error suppressing types.

The results are mechanized in Agda [1].

## 2 FURTHER WORK

TODO

## A PRAGMATIC SEMANTIC SUBTYPING

TODO

## REFERENCES


[1] L. Brown and A. S. A. Jeffrey. 2023. Luau Prototype Typechecker. https://github.com/luau-lang/agda-typeck

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

[3] A. M. Kent. 2021. Down and Dirty with Semantic Set-theoretic Types (a tutorial). https://pnwamk.github.io/sst-tutorial/

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

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

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

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

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