# Position Paper: Some Goals of the Luau Type System

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

Luau is the scripting language that powers user-generated experiences on the Roblox platform. It is a statically-typed language with type inference based on the dynamically-typed Lua language. These types are used for providing editor assistance in Roblox Studio, the IDE for authoring Roblox experiences. Due to Roblox's uniquely heterogeneous developer community, Luau must operate in a somewhat different fashion than a traditional statically-typed language. In this paper, we describe some of the goals of the Luau type system, focusing on where the goals differ from those of other type systems.

**ACM Reference Format:**
Lily Brown, Andy Friesen, and Alan Jeffrey. 2021. Position Paper: Some Goals of the Luau Type System. In *HATRA '21: Human Aspects of Types and Reasoning Assistants.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

The Roblox [15] platform allows anyone to create shared, immersive, 3D experiences. As of July 2021, there are approximately 20 million experiences available on Roblox, created by 8 million developers. Roblox creators are often young. For example, there are over 200 Roblox kids' coding camps in 65 countries listed by the company as education resources [14]. The Luau programming language [13] is the scripting language used by creators of Roblox experiences. Luau is derived from the Lua programming language [7], with additional capabilities, including a type inference engine.

This paper will discuss some of the goals of the Luau type system, such as supporting goal-driven learning, non-strict typing semantics, and mixed strict and non-strict types. Particular focus is placed on how these goals differ from traditional type systems' goals.

## 2 HUMAN ASPECTS

### 2.1 Heterogeneous developer community

Quoting a Roblox 2020 report [12]:

- Adopt Me! now has over 10 billion plays and surpassed 1.6 million concurrent users earlier this year.
- Piggy, launched in January 2020, has close to 5 billion visits in just over six months.
- There are now 345,000 developers on the platform who are monetizing their games.

This demonstrates the heterogeneity of the Roblox developer community: developers of experiences with billions of plays are on the same platform as children first learning to code. Moreover, *both of these groups are important.* The professional development studios bring high-quality experiences to the platform, and the beginning creators contribute to the energetic creative community, forming the next generation of developers.



*HATRA '21, October 2021, Chicago, IL*



ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

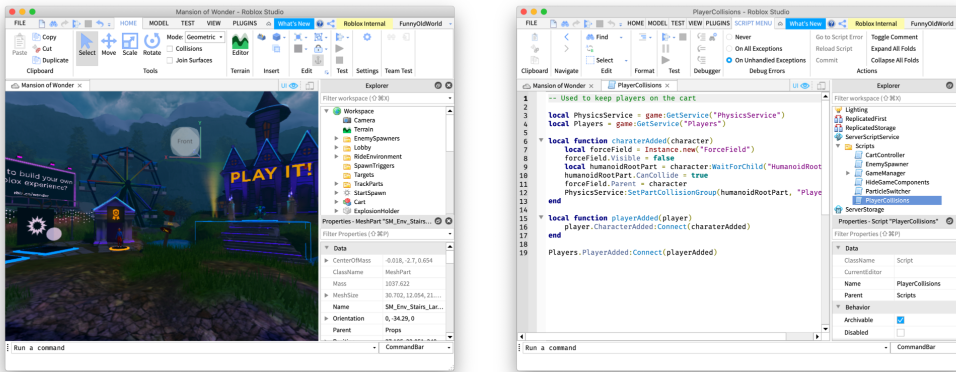


Fig. 1. Roblox Studio's 3D environment editor (a), and script editor (b)

## 2.2 Goal-driven learning

Goal: *support learning how to perform specific tasks organically*

All developers are goal-driven, but this is especially true for learners. A learner will download Roblox Studio with an experience in mind, such as designing an obstacle course (an "obby") to play in with their friends.

The user experience of developing a Roblox experience is primarily a 3D interactive one, seen in Fig. 1(a). The user designs and deploys 3D assets such as terrain, parts and joints, providing them with physics attributes such as mass and orientation. The user can interact with the experience in Studio, and deploy it to a Roblox server so anyone with the Roblox app can play it. Physics, rendering and multiplayer are all immediately accessible to creators.

At some point during experience design, the experience creator has a need which can't be met by the physics engine alone, such as "The stairs should light up when a player walks on them" or "a firework is set off every few seconds." At this point they will discover the script editor, seen in Fig. 1(b).

This onboarding experience is different from many initial exposures to programming, in that by the time the user first opens the script editor, they have already built much of their creation, and have a very specific concrete aim. As such, Luau must allow users to perform a specific task with as much help as possible from tools.

## 2.3 Type-driven development

Goal: *enable users to leverage types in their development process*

Professional development studios are also goal-directed (though the goals may be more abstract, such as "decrease user churn" or "improve frame rate") but have additional needs:

- *Code planning*: code spends much of its time in an incomplete state, with holes that will be filled in later.
- *Code refactoring*: code evolves over time, and it is easy for changes to break previously-held invariants.
- *Defect detection*: code has errors, and detecting these at runtime (for example by crash telemetry) can be expensive and recovery can be time-consuming.

Detecting defects ahead-of-time is a traditional goal of type systems, resulting in an array of techniques for establishing safety results, surveyed for example in [11]. Supporting code planning and refactoring are some of the goals of *type-driven development* [1] under the slogan "type, define, refine". For example, a common use of type-driven development is renaming a property, which is achieved by changing the name in one place, and then fixing the resulting type errors—once the type system stops reporting errors, the refactoring is complete.

To help support the transition from novice to experienced developer, types are introduced gradually, through API documentation and type discovery. Type inference provides many of the benefits of type-driven development even to creators who are not explicitly providing types.

## 3 TYPES

### 3.1 Infallible types

Goal: *provide type information even for ill-typed or syntactically invalid programs.*

Programs spend much of their time under development in an ill-typed or incomplete state, even if the final artifact is well-typed. Tools should support this by providing type information even for ill-typed or syntactically invalid programs. An analogy is infallible parsers, which perform error recovery and provide an AST for all input texts, even if they don't adhere to the parser's syntax.

Program analysis can still flag type errors, which may be presented to the user with red squiggly underlining. Formalizing this, rather than a judgment $\Gamma \vdash M : T$, for an input term $M$, there is a judgment $\Gamma \vdash M \Rightarrow N : T$ where $N$ is an output term where some subterms are *flagged* as having type errors, written $\underline{N}$. Write $\mathrm{erase}(N)$ for the result of erasing flaggings: $\mathrm{erase}(\underline{N}) = \mathrm{erase}(N)$.

The goal of infallible types is that every term can be typed:

- *Typability*: for every $M$ and $\Gamma$, there are $N$ and $T$ such that $\Gamma \vdash M \Rightarrow N : T$.
- *Erasure*: if $\Gamma \vdash M \Rightarrow N : T$ then $\mathrm{erase}(M) = \mathrm{erase}(N)$

Some issues raised by infallible types:

- Which heuristics should be used to provide types for flagged programs? For example, could one use minimal edit distance to correct for spelling mistakes in field names?
- How can we avoid cascading type errors, where a developer is faced with type errors that are artifacts of the heuristics, rather than genuine errors?
- How can the goals of an infallible type system be formalized?

*Related work*: there is a large body of work on type error reporting (see, for example, the survey in [4, Ch. 3]) and on type-directed program repair (see, for example, the survey in [8, Ch. 3]), but not type repair, or on the semantics of programs with type errors. Many compilers perform error recovery during typechecking, but do not provide a semantics for programs with type errors.

### 3.2 Strict types

Goal: *no false negatives.*

For developers who are interested in defect detection, Luau provides a *strict mode*, which acts much like a traditional, sound, type system. This has the goal of "no false negatives" where any possible run-time error is flagged. This is formalized using:

- *Operational semantics*: a reduction judgment $M \to N$ on terms.
- *Values*: a subset of terms representing a successfully completed evaluation.

Error states at runtime are represented as stuck states (terms that are not values but cannot reduce), and showing that no well-typed program is stuck. This is not true if typing is infallible, but can fairly straightforwardly be adapted. We extend the operational semantics to flagged terms, where $M \to M'$ implies $\underline{M} \to \underline{M'}$, and for any value $V$ we have $\underline{V} \to V$, then show:

- *Progress*: if $\vdash M \Rightarrow N : T$, then either $N \to N'$ or $N$ is a value or $N$ has a flagged subterm.
- *Preservation*: if $\vdash M \Rightarrow N : T$ and $N \to N'$ then $M \to^{*} M'$ and $\vdash M' \Rightarrow N' : T$.

Some issues raised by infallible types:

- How should the judgments and their metatheory be set up?
- How should type inference and generic functions be handled?
- Is the operational semantics of flagged values ($\underline{V} \rightarrow V$) the right one?
- Will higher-order code require wrappers on functions?

*Related work*: gradual typing and blame analysis, e.g. [2, 16, 17]

### 3.3 Nonstrict types

Goal: *no false positives.*

For developers who are not interested in defect detection, type-driven tools and techniques such as autocomplete, API documentation and refactoring tools can still be useful. For such developers, Luau provides a *nonstrict mode*, which we hope will eventually be useful for all developers. This non-strict typing mode is particularly useful when adopting Luau types in pre-existing code that was not authored with the type system in mind. Non-strict mode does *not* aim for soundness, but instead has the goal of "no false positives", in the sense that any flagged code is guaranteed to produce a runtime error when executed.

On the face of it, this is undecidable, since a program such as ($\mathsf{if}\ f()\ \mathsf{then\ error\ end}$) will produce a runtime error when $f()$ is true, but we can aim for a weaker property, that all flagged code is either dead code or will produce an error. Either of these is a defect, so deserves flagging, even if the tool does not know which reason applies.

We can formalize this by defining an *evaluation context* $\mathcal{E}[\bullet]$, and saying $M$ is *incorrectly flagged* if it is of the form $\mathcal{E}[\underline{V}]$. We can then define:

- *Correct flagging*: if $\vdash M \Rightarrow N : T$ then $N$ is correctly flagged.

Some issues raised by nonstrict types:

- Under this definition, any function that will terminate is unflagged, so flagging will often move from function definitions to call sites.
- This definition will not allow an unchecked use of an optional value to be flagged, for example if $f() : \mathsf{number?}$ (meaning $f$ may optionally return a number) then a strict type system can flag $1 + f()$ but a nonstrict one cannot.
- Property update of tables in languages like Luau always succeeds (the property is inserted if it did not exist), and so functions which update properties cannot be flagged.
- Does nonstrict typing require whole program analysis, to find all the possible types a property might be updated with?
- The natural formulation of function types in a nonstrict setting is that of [6]: if $f : T \rightarrow U$ and $f(V) \rightarrow^* W$ then $V : T$ and $W : U$. This formulation is *covariant* in $T$, not *contavariant*; what impact does this have?

*Related work*: success types [6] and incorrectness logic [10].

### 3.4 Mixing types

Goal: *support mixed strict/nonstrict development.*

Like every active software community, Roblox developers share code with one another constantly. First- and third-party developers alike frequently share entire software packages written in Luau. To add to this, many Roblox experiences are authored by a team. It is therefore crucial that we offer first-class support for mixing code written in strict and nonstrict modes.

Some questions raised by mixed-mode types:

- How much feedback can we offer for a nonstrict script that is importing strict-mode code?

- In strict mode, how do we talk about values and types that are drawn from nonstrict code?
- How can we combine the goals of strict and nonstrict types?
- Can we have strict and non-strict mode infer the same types, only with different flagging?
- Is strict-mode code sound when it relies on non-strict code, which has weaker invariants?

*Related work*: this appears to be an under-explored area.

### 3.5 Type inference

Goal: *infer types to allow gradual adoption of type annotations.*

Roblox, for many years, used the Lua language, which is dynamically typed and possesses a very weak type system. Due to this large quantity of pre-existing dynamically-typed code, it is essential for the type system to function even in the absence of type annotations, in order to make the type features of Luau gradually adoptable. This means that Luau needs to be able to infer types for symbols without any annotations being present, to the best of its ability. This precludes syntactical rules such as Rust's requirement that all function parameters be annotated with their type [5, Ch. 3.3].

This requirement presents challenges for the type inference algorithm, because Luau may not have enough information to determine the type of a given program. In non-strict mode in particular, which sees use in existing codebases, we cannot rely on the presence of type annotations. We also cannot require that the user provide them if Luau cannot deduce the type of a symbol. In cases such as this, we must admit defeat and assume that the code is correct, to fulfill non-strict mode's goal of "no false positives". We do this by saying that the result of the operation is `any`, a type that can be converted to and from any other type freely.

In strict mode, Luau is not so limited, and in pursuit of the strict-mode goal of "no false negatives", we may surface errors to the user indicating that the type inference system requires more information, in the form of annotations, in order to type-check a piece of code. This code, for example, requires a type annotation in order for Luau to determine the return type of the function (since Luau does not know if "+" refers to built-in addition on numbers, or a user-defined method):

```
function f(a, b)
  return a + b
end
```

Some questions raised by type inference:

- How many cases in strict mode cannot be inferred by the type inference system? Minimizing this kind of error is desirable, to make the type system as unobtrusive as possible.
- Can something like the Rust traits system [5] or Haskell classes [3] be used to provide types for overloaded operators, without hopelessly confusing learners?
- Can type inference be used to infer the same types in strict and nonstrict mode, to ease migrating between modes, with the only difference being error reporting?

*Related work*: there is a large body of work on type inference, largely summarized in [11].

## 4 CONCLUSIONS

In this paper, we have presented some of the goals of the Luau type system, and how they map to the needs of the Roblox creator community. We have also explored how these goals differ from traditional type systems, where it is necessary to accomodate the unique needs of the Roblox platform. We have sketched what a solution might look like; all that remains is to draw the owl [9].

## REFERENCES


[1] Edwin Brady. 2017. *Type-Driven Development with Idris.* Manning.

[2] Robert B. Findler and Matthias Felleisen. 2002. Contracts for Higher-order Functions. In *Proc. Int. Conf. Functional Programming*. 48–59.

[3] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (1996), 109–138.

[4] Bastiaan J. Heeren. 2005. *Top Quality Type Error Messages.* Ph.D. Dissertation. U. Utrecht.

[5] Steve Klabnik, Carol Nichols, and the Rust Community. 2021. The Rust Programming Language. https://doc.rust-lang.org/book/

[6] Tobias Lindahl and Konstantinos Sagonas. 2006. Practical Type Inference Based on Success Typings. In *Proc. Int. Conf. Principles and Practice of Declarative Programming*. 167–178.

[7] Lua.org and PUC-Rio. 2021. The Lua Programming Language. https://lua.org

[8] Bruce J. McAdam. 2002. *Repairing Type Errors in Functional Programs.* Ph.D. Dissertation. U. Edinburgh.

[9] Know Your Meme. 2010. How To Draw An Owl. https://knowyourmeme.com/memes/how-to-draw-an-owl

[10] Peter W. O'Hearn. 2020. Incorrectness Logic. In *Proc. Symp. Principles of Programming Languages.* Article 10, 32 pages.

[11] Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.

[12] Roblox. 2020. Roblox Developers Expected to Earn Over $250 Million in 2020; Platform Now Has Over 150 Million Monthly Active Users. https://corp.roblox.com/2020/07/roblox-developers-expected-earn-250-million-2020-platform-now-150-million-monthly-active-users/

[13] Roblox. 2021. The Luau Programming Language. https://luau-lang.org

[14] Roblox. 2021. Roblox Education: All Educators. https://education.roblox.com/en-us/educators

[15] Roblox. 2021. What is Roblox. https://corp.roblox.com

[16] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proc. Scheme and Functional Programming Workshop*. 81–92.

[17] Philip Wadler and Robert B. Findler. 2009. Well-typed Programs Can't be Blamed. In *Proc. European Symp. Programming.* 1–16.