Position Paper: Some Goals of the Luau Type System

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

Luau is the scripting language used in creating Roblox experiences. It is a statically-typed language based on the dynamically-typed Lua language, and uses type inference to infer types. These types are used in the IDE, for example when providing autocomplete suggestions. In this paper, we describe some of the goals of the Luau type system, focusing on where the goals are different 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, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Roblox [13] platform allows anyone to create shared, immersive, 3D experiences. At the time of writing, 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 at [12]. The Luau programming language [11] is the scripting language used by developers of Roblox experiences. Luau is derived from the Lua programming language [4], with additional capabilities, including a type inference engine.

This paper will discuss some of the goals of the Luau type system, focusing on where the goals are different from those of other type systems.

2 HUMAN ASPECTS

2.1 Heterogeneous developer community

Quoting a Roblox 2020 report [10]:

- Adopt Me! now has over 10 billion plays and surpassed 1.6 million concurrent users in game 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.

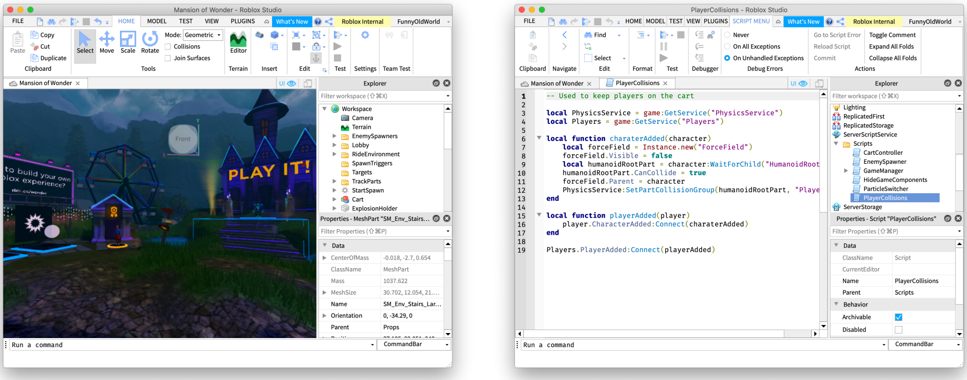
This demonstrates how heterogeneous the Roblox developer community is: developers of experiences with billions of plays are on the same platform as children first learning to code. Moreover, *both of these groups are important*, as the professional development studios bring high-quality experiences to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

2.2 Goal-driven learning

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

HATRA '21, October 2021, Chicago, IL

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



```
-- Used to keep players on the cart

local PhysicsService = game:GetService("PhysicsService")
local Players = game:GetService("Players")

local function charaterAdded(character)
    local forceField = Instance.new("ForceField")
    forceField.Visible = false
    local humanoidRootPart = character:WaitForChild("HumanoidRoot
    humanoidRootPart.CanCollide = true
    forceField.Parent = character
    PhysicsService:SetPartCollisionGroup(humanoidRootPart, "Playe
end

local function playerAdded(player)
    player.CharacterAdded:Connect(charaterAdded)
end

Players.PlayerAdded:Connect(playerAdded)
```

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

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, and provides 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.

At some point during experience design, the user of Studio has a need which can't be met by the physics engine alone. "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), and the Luau programming language.

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. It suggests a Luau goal for helping the majority of creators: *support learning how to perform specific tasks* (for example through autocomplete suggestions).

2.3 Type-driven development

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 development time in an incomplete state, with holes that will be filled in later.
- *Code refactoring*: experiences evolve over time, and it 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 [8]. 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 to rename 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: *support type-driven tools for all programs.*

Programs spend much of their time under development in an incomplete state, even if the final artifact is well-typed. Tools should support this, by providing type information for all programs. An analogy is infallible parsers, which perform error recovery and provide an AST for all input texts.

Program analysis can still flag type errors, for example 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 Γ, 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 [2, Ch. 3]) and on type-directed program repair (see, for example, the survey in [5, 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" that is any run-time error is flagged. This is formalized using:

- *Operational semantics*: a reduction judgment $M \longrightarrow N$ 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 $M \longrightarrow M'$ implies $\underline{M} \longrightarrow \underline{M'}$, and for any value V we have $\underline{V} \longrightarrow V$, then show:

- *Progress*: if $\vdash M \Rightarrow N : T$, then either $N \longrightarrow N'$ or N is a value or N has a flagged subterm.
- *Preservation*: if $\vdash M \Rightarrow N : T$ and $N \longrightarrow N'$ then $M \longrightarrow^* M'$ 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 ($\underline{V} \longrightarrow V$) the right one?
- Will higher-order code require wrappers on functions?

Related work: gradual typing and blame analysis, e.g. [9, 14, 15]

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 support for refactoring can still be useful. For such developers, Luau provides a *nonstrict mode*, which we hope will eventually be useful for all developers. This 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 $(\mathsf{if}\ f()\ \mathsf{then}\ \mathsf{error}\ \mathsf{end})$ 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 [**?**]: 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 [3] and incorrectness logic [7].

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 games are authored not by just one developer, but a team. It is therefore crucial that we offer first-class support for mixing code written in strict and nonstrict modes.

Some issues 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?

Related work: this appears to be an undererxplored area.

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 sketched what a solution might look like, all that remains is to draw the owl [6].

REFERENCES

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

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

[3] 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.

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

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

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

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

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

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

[10] 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/

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

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

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

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

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