HATRA 2023

Goals of the Luau Type System, Two Years On

Lily Brown, Andy Friesen, and Alan Jeffrey

ROBLOX

CC BY

Table of Contents

1. Recap: HATRA 2021
2. Progress: semantic subtyping
3. Progress: gradual typing
4. Future work

Recap: HATRA 2021

https://asaj.org/papers/hatra21.pdf

Position Paper: 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, based on the dynamically-typed Lua language, with type inference. 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: Goals of the Luau Type System. In *HATRA '21: Human Aspects of Types and Reasoning Assistants.* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

The Roblox 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 [19]. Roblox creators are often young: there are over 200 Roblox kids' coding camps in 65 countries listed by the company as education resources [18]. The Luau programming language [17] 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

What is Roblox?

Roblox is **not** a games company!

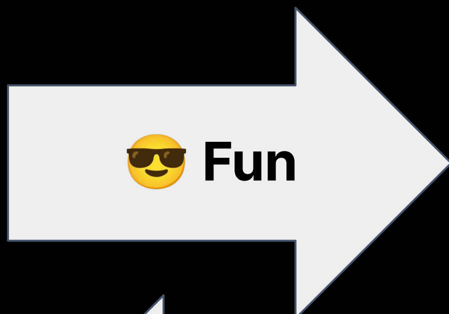
What is Roblox?

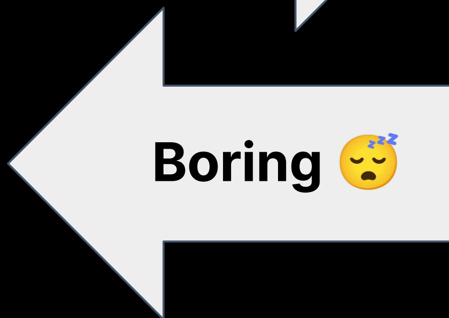
Roblox is a **platform** for shared virtual experiences

What is Roblox?

Roblox is a **platform** for shared virtual experiences

Roblox provides
Physics engine
Rendering pipeline
Multimedia
Distribution
Creation tools
X-platform clients
Programmability
Trust & safety
Security
Community
Economy
Avatars
Digital matter
...



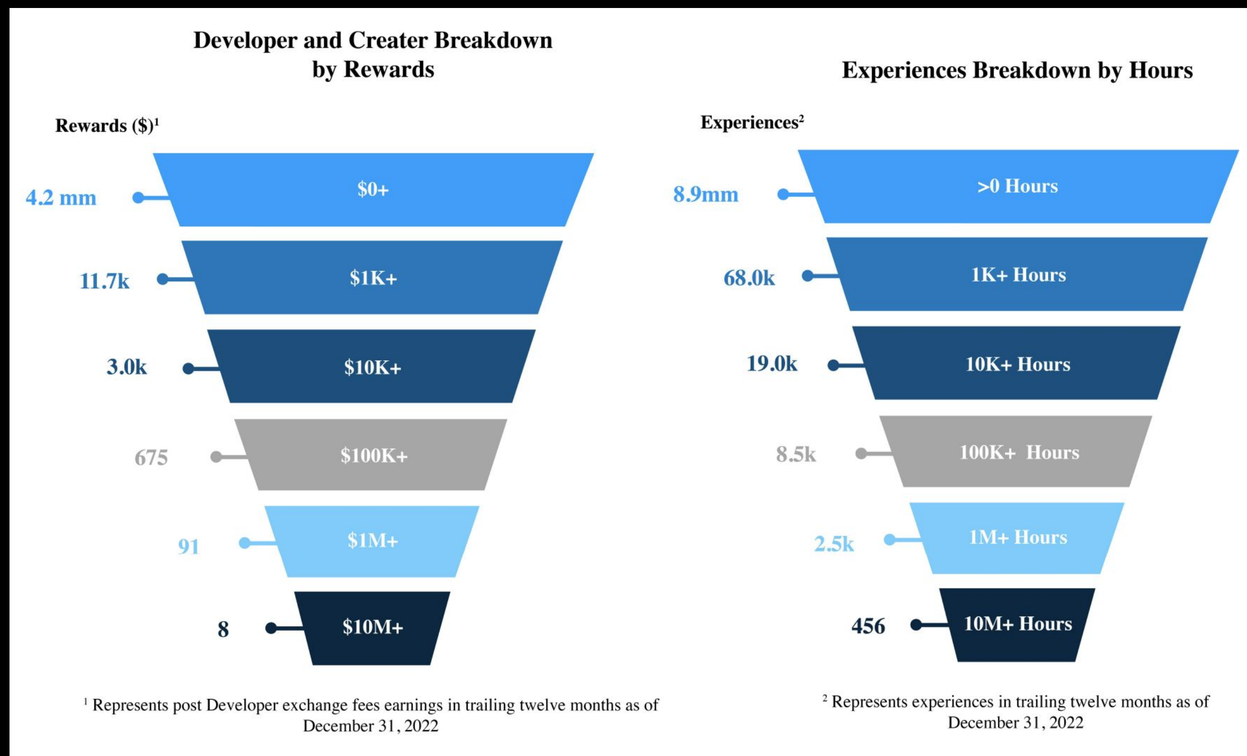


Creators provide
Creativity
Design
Storytelling
Music
Laughter
Thrills
...

Who are Roblox creators?

Very heterogeneous creator community

- Most creators (measured by $) are professional devs
- Most creators (measured by people) are kids having fun
- Both are important!



https://ir.roblox.com/financials/annual-reports/

Who are Roblox creators?

Different creators have different needs

- Professional developers have code quality goals
- Beginning creators have immediate goals
- For both, type-driven productivity tools (autocomplete, API docs, ...) are important

Technical impact of heterogeneous creator community

Correlation of run-time errors and static type errors

Predicting run-time errors is undecidable, so you have a design decision:

- **Sound** systems have **no false negatives**: if a program type-checks, then it has no run-time errors
- **Complete** systems have **no false positives**: if a program has a type error, then it has a run-time error

Traditional POPL-style type systems are sound, complete systems do exist (incorrectness logic, Elixir, …) but are rarer.

Luau supports both via **strict mode** and **non-strict mode**.

Progress: semantic subtyping

What is semantic subtyping?

Subtyping “the way you first think it works”

- Types are interpreted as sets of values
- Subtyping is interpreted as subset inclusion

D’oh!

What is semantic subtyping?

Not quite as easy as you might think

https://www.irif.fr/~gc/papers/icalp-ppdp05.pdf

close the circle. The algorithm to compute the subtyping relation, however, does depend on the choice of the model. The model $\mathscr{U}$ is a natural choice because it is universal, in the sense that it induces …ping relation; formally, for every model

$$\ldots_{\mathscr{D}} t_2 \;\Rightarrow\; t_1 \leq_{\mathscr{U}} t_2$$

…oo poor a structure to be used as the target …s programming language, as it cannot ex… Nevertheless it is enough for interpreting …terising type containment).

…ntation and subtyping

…mantically defined subtyping relation the …f an effective way to check whether two … relation. In order to define the subtyping … easier to work with types that are written …ng a canonical form is not very hard since …semantic interpretation of types. So let us …hat can be seen as the set of regular trees

$$\ldots \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

…row type $t \to t$ or a product type $t \times t$. A …*nal form* if and only if it is a finite union …toms or their negations (in the latter case …*ns*). For instance:

$$\ldots_4 \wedge \neg a_5) \vee (\neg a_6 \wedge \neg a_7) \vee (a_8 \wedge a_9)$$

…e identify $\mathbf{0}$ with the empty union and $\mathbf{1}$ …n).

… and t are *equivalent* if they have the same … $= [\![t]\!]$) and we denote it by $s \simeq t$. Every …uivalent to a type in disjunctive normal … loss of generality we can consider this … of types we work on.

…ur representation by noting that if the in…ms with different constructors, then they … to a unique type according to the polari…: so for instance $(s_1 \times t_1) \wedge (s_2 \to t_2) \simeq \mathbf{0}$ … $\simeq s_1 \times t_1$. Therefore we only consider … atoms of the same sort (that is, that do …v types), e.g. :

$$\ldots \wedge (s_2 \times t_2) \wedge \neg(s_3 \times t_3))$$

$(\{(s_1 \times t_1), (s_2 \times t_2)\}, \{(s_3 \times t_3)\})$. Therefore every packet can be represented by a set S of such pairs, under the following form:

$$\bigvee_{(P,N)\in S} \Big((\bigwedge_{a\in P} a) \wedge (\bigwedge_{a\in N} \neg a) \Big)$$

Thus two of such sets are all we need to represent every type. For instance our previous type is represented by the pair

$$\left(\left\{ \begin{array}{c} (\{(s_1{\times}t_1), (s_2{\times}t_2)\}, \{(s_3{\times}t_3)\}) \\ (\{(s_2 \times t_3), (s_4 \times t_1)\}, \{\}) \end{array} \right\}, \{(\{\}, \{(s_4 \to t_4), (s_5 \to t_5)\})\} \right)$$

It is interesting to notice that this is not just a theoretical representation but it is also the representation we used in early versions of the $\mathbb{C}$Duce interpreter (the current implementation is using more efficient partial decision trees).

But let us go back to our problem, which is the one of defining algorithms that verify whether two types s and t are in the subtyping relation. The key observation for what follows is again that the problem of deciding whether two types are in subtyping relation can be reduced to the problem of deciding whether a type is empty. Recall

$$s \leq t \iff [\![s]\!] \subseteq [\![t]\!] \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s \wedge \neg t]\!] = \varnothing.$$

Since $\varnothing = [\![\mathbf{0}]\!]$ then

$$s \leq t \iff s \wedge \neg t \simeq \mathbf{0}$$

Since every type can be represented as the union of addenda of uniform sort and a union is empty only if all its addenda are empty, then in order to decide the emptiness of every type it suffices to establish when the terms

$$A = (\bigwedge_{a\in P} a) \wedge (\bigwedge_{a\in N} \neg a)$$

are empty for P and N formed by types of the same sort (all products or all arrows). Or equivalently this results to deciding

$$(\bigwedge_{a\in P} a) \leq (\bigvee_{a\in N} a)$$

that is, we must be able to decide whether

$$(\bigwedge_{s\times t\in P} s \times t) \wedge (\bigwedge_{s\times t\in N} \neg(s \times t)) \qquad (2)$$

and

$$(\bigwedge_{s\to t\in P} s \to t) \wedge (\bigwedge_{s\to t\in N} \neg(s \to t)) \qquad (3)$$

are equivalent to $\mathbf{0}$. So the algorithm must decompose this problem …

A Gentle Introduction to Semantic Subtyping

Giuseppe Castagna
CNRS
École Normale Supérieure
Paris, France

Alain Frisch
INRIA
Rocquencourt
France

ABSTRACT

Subtyping relations are usually defined either syntactically by a formal system or semantically by an interpretation of types into an untyped denotational model. In this work we show step by step how to define a subtyping relation semantically in the presence of functional types and dynamic dispatch on types, without the complexity of denotational models, and how to derive a complete subtyping algorithm. It also provides a recipe to add set-theoretic union, intersection, and negation types to your favourite language.

The presentation is voluntarily kept informal and discursive and the technical details are reduced to a minimum since we rather insist on the motivations, the intuition, and the guidelines to apply the approach.

Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features — Data types and structure; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs — Type structure; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic — Lambda calculus and related systems

General Terms: Language, Theory.

Keywords: Typing; Subtyping; Intersection, Union, and Negation Types.

in which types can be interpreted as subsets of a model may be a hard task. A solution to this problem was given by Haruo Hosoya and Benjamin Pierce [18, 17, 16] with the work on XDuce. The key idea is that in order to define the subtyping relation semantically one does not need to start from a model of the whole language: a model of the types suffices. In particular Hosoya and Pierce take as model of types the set of values of the language. Their notion of model cannot capture functional values. On the one hand, the resulting type system is poor since it lacks function types. On the other hand, it manages to integrate union, product and recursive types and still keep the presentation of the subtyping relation and of the whole type system quite simple.

In [12, 11], together with Véronique Benzaken, we extended the work on XDuce and reframed it in a more general setting: we show a technique to define semantic subtyping in the presence of a rich type system including function types, but also arbitrary boolean combinations (union, intersection, and negation types) and in the presence of lately bound overloaded functions and type-based pattern matching. The aim of [12, 11] was to provide a theoretical foundation on the top of which to build the language $\mathbb{C}$Duce [6], an XML-oriented transformation language. This motivation needed a rather heavy technical development that concealed a side—but important—contribution of the work, namely a generic and uniform technique (or rather, a cookbook of techniques) to define se…

Why semantic subtyping?

Minimizing false positives

ROBLOX About Play

Horrible mess of overloaded functions and union types, resulting in a false positive.

One of the sources of false positives in Luau (and many other similar languages like TypeScript or Flow) is *subtyping*. Subtyping is used whenever a variable is initialized or assigned to, and whenever a function is called: the type system checks that the type of the expression is a subtype of the type of the variable. For example, if we add types to the above program

```
local x : CFrame = CFrame.new()
local y : Vector3 | CFrame
if math.random() < 0.5 then
  y = CFrame.new()
else
  y = Vector3.new()
end
local z : Vector3 | CFrame = x * y
```

then the type system checks that the type of `CFrame` multiplication is a subtype of `(CFrame, Vector3 | CFrame) -> (Vector3 | CFrame)`.

Subtyping is a very useful feature, and it supports rich type constructs like type union (`T | U`) and intersection (

Implementation

Not subtyping the way you first think it works

- First, try syntactic subtyping, if that succeeds, then yay!
- If not, perform **type normalization** (with possible exponential blowup due to CNF)
- Semantic and syntactic subtyping coincide on normalized types.

Lots of devils in the details.

Progress: gradual typing

Traditional gradual types

Compatible types (aka what we used to do)

- Add a type **any**, allow it to be used wherever
- Define “compatibility”, in particular $T \sim$ **any** $\sim U$
- Use compatible typing everywhere, e.g. if $f : F$ and $x : T$ and $F \sim T \rightarrow U$ then $f(x) : U$

Easy to get wrong, since ~ isn’t transitive.

Error suppression

Treat type warnings constructively (aka what we do now)

- Give every term-in-context a type typeof(Γ,M),
  e.g. typeof(Γ, *f*(*x*)) is apply(typeof(Γ,*f*), typeof(Γ,*x*)).
- Define which typings generate type warnings,
  e.g. function application generates a warning when
  typeof(*x*) $\not<:$ src(typeof(*f*)), and **there are no error-suppressing types!**
- Needs apply($T \to U$, V) is U, src($T \to U$) is T, and **any** is error-suppressing.

Error suppression

Constructive formulation of type soundness for strict mode

Two readings:

- constructive statement of “well typed programs don’t go wrong”
- run a program, if it produces a run-time error, then run it back in time (!) and find a root cause type warning

Time travel debugging for type systems!

```
wellTypedProgramsDontGoWrong : ∀ H′ B B′ → (∅ᴴ ⊢ B ⟶* B′ ⊣ H′) → (RuntimeErrorᴮ H′ B′) → Warningᴮ ∅ᴴ (typeCheckᴮ ∅ᴴ ∅ B)
wellTypedProgramsDontGoWrong H′ B B′ t err with reflect* ∅ᴴ B t (runtimeWarningᴮ H′ B′ err)
wellTypedProgramsDontGoWrong H′ B B′ t err | heap (addr a refl ())
wellTypedProgramsDontGoWrong H′ B B′ t err | ctxt (UnsafeVar x () p)
wellTypedProgramsDontGoWrong H′ B B′ t err | block W = W
```

Future work

Future work

Drawing more owls

- Flesh out non-strict mode
- Treat non-strict mode as “strict mode, but with more error suppression”
- Actually ship

Thank you!