Goals of the Luau Type System, Two Years On

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

In HATRA 2021, we presented *The Goals Of The Luau Type System*, describing the human factors of a type system for a language with a heterogeneous developer community. In this extended abstract we provide a progress report, focusing on the unexpected aspects: semantic subtyping and type error suppression.

ACM Reference Format:
Lily Brown, Andy Friesen, and Alan Jeffrey. 2023. Goals of the Luau Type System, Two Years On. In *HATRA '23: Human Aspects of Types and Reasoning Assistants.* ACM, New York, NY, USA, 2 pages.

1 RECAP

Luau [7] is the scripting language used by the Roblox [8] platform for shared immersive experiences. Luau extends the Lua [5] language, notably by providing type-driven tooling such as autocomplete and API documentation (as well as traditional type error reporting). Roblox has hundreds of millions of users, and millions of creators, ranging from children learning to program for the first time to professional development studios.

In HATRA 2021, we presented a position paper on the *Goals Of The Luau Type System* [1], describing the human factors issues with designing a type system for a language with a heterogeneous developer community. The design flows from the needs of the different communities: beginners are focused on immediate goals ("the stairs should light up when a player walks on them") and less on the code quality concerns of more experienced developers; for all users type-driven tooling is important for productivity. These needs result in a design with two modes:

- *non-strict mode*, aimed at non-professionals, focused on minimizing false positives (that is, in non-strict mode, any program with a type error has a defect), and
- *strict mode*, aimed at professionals, focused on minimizing false negatives (that is, in strict mode, any program with a defect has a type error).

2 PROGRESS

In the two years since the position paper, we have been making changes to the Luau type system to achieve the goals we set out. Most of the changes were straightforward, but two were large changes in how we thought about the design of the type system: replacing the existing syntactic subtyping algorithm by *semantic subtyping*, and treating gradual typing as *type error suppression.*

Semantic subtyping interprets types as sets of values, and subtyping as set inclusion [3]. This is aligned with the *minimize false positives* goal of Luau non-strict mode, since semantic subtyping only reports a failure of subtyping when there is a value which inhabits the candidate subtype, but not the candidate supertype. For example, the 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
```

cannot produce a run-time error, since multiplication of `CFrame`s is overloaded:

```
((CFrame, CFrame) -> CFrame) & ((CFrame, Vector3) -> Vector3)
```


HATRA '23, October 2023, Portugal, Spain

In order to typecheck this program, we check that that type is a subtype of:

```
(CFrame, Vector3 | CFrame) -> (Vector3 | CFrame)
```

In the previous, syntax-driven, implementation of subtyping, this subtype check would fail, resulting in a false positive. We have now released an implementation of semantic subtyping, which does not suffer from this defect. See our technical blog for more details [4].

Rather than the gradual typing approach of Siek and Taha [9], which uses *consistent subtyping* where $\texttt{any} \lesssim T \lesssim \texttt{any}$ for any type T, we adopt an approach based on *error suppression*, where any is an error-suppressing type, and any failures of subtyping involving error-suppressing types are not reported. Users can explicitly suppress type errors by declaring variables with type any, and since an expression with a type error has an error-suppressing type we avoid cascading errors.

We do this by defining a *infallible* typing judgment $\Gamma \vdash M : T$ such that for any Γ and M, there is a T such that $\Gamma \vdash M : T$. For example the rule for addition (ignoring overloads for simplicity) is:

$$\frac{\Gamma \vdash M : T \quad \Gamma \vdash M : U}{\Gamma \vdash M + N : \texttt{number}}$$

We define which judgments produce warnings, for example that rule produces a warning when

- either $T \not<: \texttt{number}$ and T is not error-suppressing,
- or $U \not<: \texttt{number}$ and U is not error-suppressing.

To retain type soundness (in the absence of user-supplied error-suppressing types) we show that if $\Gamma \vdash M : T$ and T is error-suppressing, then either

- Γ or M contains an error-suppressing type, or
- $\Gamma \vdash M : T$ produces a warning.

From this it is straightforward to show the usual "well typed programs don't go wrong" type soundness result for programs without explicit error-suppressing types [2].

3 FURTHER WORK

Currently the type inference system uses greedy inference, which is very sensitive to order of declarations, and can easily result in false positives. We plan to replace this by some form of local type inference [6].

Currently, non-strict mode operates in the style of gradual type systems by inferring any as the type for local variables. This does not play well with type-directed tooling, for example any cannot provide autocomplete suggestions. Local type inference will infer more precise union types, and hence better type-driven tooling.

At some point, we hope that error suppression will be the only difference between strict mode and non-strict mode.

REFERENCES

[1] L. Brown, A. Friesen, and A. S. A. Jeffrey. 2021. Position Paper: Goals of the Luau Type System. In *Proc. Human Aspects of Types and Reasoning Assistants*. https://asaj.org/papers/hatra21.pdf

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

[3] G. Castagna and A. Frisch. 2005. A Gentle Introduction to Semantic Subtyping. In *Proc. Principles and Practice of Declarative Programming*.

[4] A. S. A. Jeffrey. 2022. Semantic Subtyping in Luau. Roblox Technical Blog. https://blog.roblox.com/2022/11/semantic-subtyping-luau/

[5] Lua.org and PUC-Rio. 2023. The Lua Programming Language. https://lua.org

[6] B. C. Pierce and D. N. Turner. 2000. Local Type Inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44.

[7] Roblox. 2023. The Luau Programming Language. https://luau-lang.org

[8] Roblox. 2023. Reimagining the way people come together. https://corp.roblox.com

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