Towards an Unsound But Complete Type System

Work In Progress on New Non-Strict Mode for Luau

Lily Brown
Andy Friesen
Alan Jeffrey
Vighnesh Vijay
Roblox
San Mateo, CA, USA

Abstract

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. One of the goals was the design of type system for bug detection, where we have high confidence that type errors isolate genuine software defects, and that false positives are minimized. Such a type system is, by necessity, unsound, but we can ask instead that it is complete. This paper presents a work-in-progress report on the design and implementation of the new non-strict mode for Luau.
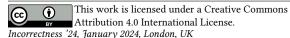
1 Introduction

Luau [6] is the scripting language used by the Roblox [7] 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).

The focus of this extended abstract is the design of non-strict mode: what constitutes a defect, and how can we design a complete type system for detecting them. (The words "sound" and "complete" in this sense arte far from ideal, but "sound type system" has a well-established meaning, and "complete" is well-established as the dual of "sound", so here we are.)

The closest work to ours is success typing [4], used in Erlang Dialyzer [3]. The new feature of our work is that strict and non-strict mode have to intertact, whereas Dialyzer only has the equivalent of non-strict mode.

New non-strict mode is specified in a Luau Request For Comment [2], and is currently being implemented. We expect it (and other new typechecking features) to be available in 2024. This extended abstract is based on the RFC, but written in "greek letters and horizontal lines" rather than "monspaced text" for mat.

2 Code defects

The main goal of non-strict mode is to identify defects, but this requires identifying what constitutes a defect,. Run-time errors are an obvious defect:

```
local hi = "hi"
print(math.abs(hi))
```

but we also want to catch common mistakes such as misspelling a property name, even though Luau returns `nil` for missing property accesses. For this reason, we consider a larger class of defects:

- run-time errors,
- expressions guaranteed to be `nil`, and
- writing to a table property that is never read.

2.1 Run-time errors

Run-time errors occur due to run-time type mismatches (such as 5("hi")) or buit-in functios (such as math.abs("hi")). Detecting run-time errors is undecidable, for example:

```
if cond() then
  math.abs("hi")
end
```

In general, we cannot be sure that this code produces a run-time error, but we do know that if math.abs("hi") is executed, it will produce an error, so we consider this to be a defect.

2.2 Expressions guaranteed to be nil

Luau tables do not error when a missing property is accessed (though embeddings may). So

```
local t = { Foo = 5 }
local x = t.Fop
```

won't produce a run-time error, but is more likely than not a programmer error. In this case, if the programmer intent was to initialize x as nil, they could have initialized x = nil. For this reason, we consider it a code defect to use an expression that the type system infers is of type nil, other than the nil constant itself.

2.3 Writing properties that are never read

There is a matching problem with misspelling properties when writing. For example

```
function f()
  local t = {}
  t.Foo = 5
  t.Fop = 7
  print(t.Foo)
end
```

won't produce a run-time error, but is more likely than not a programmer error, since t.Fop is written but never read. We can use read-only and write-only table properties types for this, and make it an error to create a write-only property.

We have to be careful about this though, because if f ended with return t, then it would be a perfectly sensible function with type () -> { Foo: number, Fop: number }. The only way to detect that Fop was never read would be whole-program analysis, which is prohibitively expensive.

3 New Non-strict error reporting

The difficult part of non-strict mode error-reporting is detecting guaranteed run-time errors. We do this using an error-reporting pass that generates a type context such that if any of the $x : T$ in the type context are satisfied, then the program is guaranteed to produce a type error.

For example in the program

```
function h(x, y)
  math.abs(x)
```

```
    string.lower(y)
  end
```

an error is reported when x isn't a number, or y isn't a string, so the generated context is

```
x : ~number
y : ~string
```

(~T is Luau's concrete syntax for type negation.) In the function:

```
function f(x)
  math.abs(x)
  string.lower(x)
end
```

an error is reported when x isn't a number or isn't a string, so the constraint set is

```
x : ~number | ~string
```

(T|U is Luau's concrete syntax for type negation.) Since ~number | ~string is equivalent to the type unknown (which contains every value), non-strict mode can report a warning, since calling the function is guaranteed to throw a run-time error. In contrast:

```
function g(x)
  if cond() then
    math.abs(x)
  else
    string.lower(x)
  end
end
```

generates context

```
x : ~number & ~string
```

(T&U is Luau's concrete syntax for type intersection.) Since ~number & ~string is not equivalent to unknown, non-strict mode reports no warning.

In Figure reffig:ctxtgen we provide some of the inference rules for conbtext generation, and the warnings that context generation produces. These are run after type inference, so theay can assume that all code is fully typed.

These rules generararalize type intersection and union to type contexts, write $\emptyset$ for the everywhere-never context, and write $x : T$ for the context that is everywhere never except for x where it maps to T.

Conjecture 3.1. *If* $\Gamma \vdash M : T \dashv \Delta, x : U$ *and* σ *is a closing substitution where* $\sigma(x) : U$, *and* $M[\sigma] \rightarrow^* v$ *then* $v : T$.

Corollary 3.2. *If* $\Gamma \vdash M :$ never $\dashv \Delta, x :$ unknown *and* σ *is a closing substitution and* $M[\sigma]$ *does not terminate successfully.*

4 Checked functions

TODO

5 Future work

TODO

$$\dfrac{\begin{array}{c}\Gamma \vdash M : \mathsf{never} \dashv \Delta_1 \\ \Gamma, x : T \vdash B \dashv \Delta_2, x : U \\ (\text{warn if } \mathsf{unknown} <: U)\end{array}}{\Gamma \vdash (\mathsf{local}\, x : T = M; B) \dashv (\Delta_1 \cup \Delta_2)} \qquad \dfrac{\begin{array}{c}\Gamma \vdash M : \mathsf{never} \dashv \Delta_1 \\ \Gamma \vdash B \dashv \Delta_2 \\ \Gamma \vdash C \dashv \Delta_3\end{array}}{\Gamma \vdash (\mathsf{if}\, M \,\mathsf{then}\, B \,\mathsf{else}\, C \,\mathsf{end}) \dashv (\Delta_1 \cup (\Delta_2 \cap \Delta_3))}$$

$$\dfrac{}{\Gamma \vdash x : T \dashv (x : T)} \qquad \dfrac{(\text{warn if } k : T)}{\Gamma \vdash k : T \dashv \emptyset} \qquad \dfrac{\begin{array}{c}\Gamma, x : S \vdash B \dashv \Delta, x : U \\ (\text{warn if } \mathsf{unknown} <: U) \\ (\text{warn if } \mathsf{function} <: T)\end{array}}{\Gamma \vdash (\mathsf{function}(x : S)B\,\mathsf{end}) : T \dashv \Delta} \qquad \dfrac{\begin{array}{c}\Gamma \vdash M : (S \to \mathsf{error}) \\ \Gamma \vdash M : \neg\mathsf{function} \dashv \Delta_1 \\ \Gamma \vdash N : S \dashv \Delta_2 \\ (\text{warn if } \Gamma \vdash M : (\mathsf{unknown} \to T))\end{array}}{\Gamma \vdash M(N) : T \dashv \Delta_1 \cup \Delta_2}$$

Figure 1. Type context generation for blocks ($\Gamma \vdash B \dashv \Delta$) and expressions ($\Gamma \vdash M : T \dashv \Delta$)

$$\dfrac{}{\Gamma \vdash \mathsf{math.abs} : (\neg\mathsf{number} \to \mathsf{error})} \quad \dfrac{}{\Gamma \vdash \mathsf{math.abs} : \neg\mathsf{function} \dashv \emptyset} \quad \dfrac{\begin{array}{c}\Gamma \vdash \mathsf{string.lower} : (\neg\mathsf{string} \to \mathsf{error}) \\ \Gamma \vdash \mathsf{string.lower} : \neg\mathsf{function} \dashv \emptyset \\ \Gamma \vdash x : \neg\mathsf{string} \dashv (x : \neg\mathsf{string}) \\ (\text{warn since } \Gamma \vdash \mathsf{string.lower} : \mathsf{unknown} \to \neg\mathsf{number})\end{array}}{\Gamma \vdash \mathsf{string.lower}(x) : \neg\mathsf{number} \dashv (x : \neg\mathsf{string})}$$
$$\overline{\Gamma \vdash (\mathsf{math.abs}(\mathsf{string.lower}(x)) : \mathsf{never} \dashv (x : \neg\mathsf{string})}$$

Figure 2. Example warning

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] Alan Jeffrey. 2023. RFC For Newv Non-strict Mode. Luau Request For Comment. https://github.com/Roblox/luau/pull/1037.

[3] Tobias Lindahl and Konstantinos Sagonas. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In *Proc. Asian Symp. Programming Languages and Systems*. 91–106.

[4] 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.

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

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

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