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 identify 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 unsound type system for Luau.

ACM Reference Format:
Lily Brown, Andy Friesen, Alan Jeffrey, and Vighnesh Vijay. 2024. Towards an Unsound But Complete Type System: Work In Progress on New Non-Strict Mode for Luau. In *Incorrectness '24: Formal Methods for Incorrectness.* ACM, New York, NY, USA, 4 pages.

1 Introduction

Luau [12] is the scripting language used by the Roblox [13] platform for shared immersive experiences. Luau extends the Lua [10] 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, which minimizes false positives (that is, in non-strict mode, any program with a type error has a defect), and
- *strict mode*, aimed at professionals, which minimizes 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 are 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 [9], used in Erlang Dialyzer [8]. The new feature of our work is that strict and non-strict mode have to interact, whereas Dialyzer only has the equivalent of non-strict mode.

New non-strict mode is specified in a Luau Request For Comment [7], and is currently being implemented. We expect it (and other new type checking features) to be available in 2024. This extended abstract is based on the RFC, but written in "Greek letters and horizontal lines" rather than "monospaced text".

2 Code defects

The main goal of non-strict mode is to identify defects, but this requires deciding what a defect is. 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 incorrect built-in function calls (such

Incorrectness '24, January 2024, London, UK

as `math.abs("hi")`). Precisely identifying run-time errors is undecidable, for example:

```
if cond() then
  math.abs(“hi”)
end
```

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
```

does not produce a run-time error, but is more likely than not a programmer error. If the programmer intended to initialize `x` as `nil`, they could have written `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` literal.

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
```

does not 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 consider it an code defect 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 predicting run-time errors. We do this using an error-reporting pass that synthesizes a type context Δ such that if any of the $x : T$ in Δ are satisfied, then the program will 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 synthesized context is

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

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

```
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 context is

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

(`T | U` is Luau's concrete syntax for type union.) Since the type `~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
```

synthesizes 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.

This use of intersection and union types is similar to the treatment of angelic and demonic nondeterminism in [**?**].

In Figure 1 we provide some of the inference rules for context synthesis, and the warnings that it produces. These are run after type inference, so they can assume that all code is fully typed.

In the judgment $\Gamma \vdash M : T \dashv \Delta$, the type context Γ is the usual *checked* type context and Δ is the *synthesized* context used to predict run-time errors (following the terminology of bidirectional typing [5]).

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

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

4 Checked functions

The crucial aspect of this type system is that we have a type error inhabited by no values, and by expressions which may throw a run-time exception. (This is essentially a very simple type and effect system [11] with one effect.)

The rule for function application $M(N)$ has two dependencies on the type for M:

$$\Gamma \vdash M : (S \to \mathsf{error})$$
$$\Gamma \vdash M : (\mathsf{unknown} \to (T \cup \mathsf{error}))$$

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

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

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

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

Figure 2. Example warning

Since Luau is based on semantic subtyping [4, 6] and supports intersection types, this is equivalent to asking for M to be an overloaded function, where one overload has argument type unknown, and one has result type error. For example:

$$\mathsf{math.abs} : (\mathsf{number} \to \mathsf{number}) \cap (\neg\mathsf{number} \to \mathsf{error})$$

and so (by subsumption):

$$\begin{gathered}\mathsf{math.abs} : (\neg\mathsf{number} \to \mathsf{error}) \\ \mathsf{math.abs} : (\mathsf{unknown} \to (\mathsf{number} \cup \mathsf{error}))\end{gathered}$$

This is a common enough idiom it is worth naming it: we call any function of type

$$(S \to T) \cap (\neg S \to \mathsf{error})$$

a *checked* function, since it performs a run-time check on its argument. They are called *strong arrows* in Elixir [3].

Note that this type system has the usual subtyping rule for functions: contravariant in their argument type, and covariant in their result type. In contrast, checked functions are invariant in their argument type, since one overload $S \to T$ is contravariant in S, and the other $\neg S \to \mathsf{error}$ is covariant.

This system is also different from success typings [9], which has functions $(\neg S \to \mathsf{error}) \cap (\mathsf{unknown} \to (T \cup \mathsf{error}))$, in our system, which are covariant in both S and T.

This is also similar to *necessity arrows* from [**?**]. Their type system is similar to ours in that both systems have typing contexts on both the left and right-hand side of a typing judgement.

5 Future work

This type system is still in the design phase [7], though we hope the implementation will be ready by the end of 2023. This will include testing the implementation on our unit tests, and on large code bases.

There is an Agda development of a core of strict mode [2]. It should extend to non-strict mode, at which point Conjecture 3.1 (or something like it) will be mechanically verified.

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, G. Duboc, and J. Valim. 2023. The Design Principles of the Elixir Type System. https://doi.org/10.48550/arXiv.2306.06391.

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

[5] J. Dunfield and N. Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5, Article 98 (2022), 38 pages.

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

[7] A. S. A. Jeffrey. 2023. RFC For New Non-strict Mode. Luau Request For Comment. https://github.com/Roblox/luau/pull/1037.

[8] T. Lindahl and K. 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.

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

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

[11] F. Nielson and H. R. Nielson. 1999. *Type and Effect Systems*. Springer, 114–136.
[12] Roblox. 2023. The Luau Programming Language. https://luau-lang.org
[13] Roblox. 2023. Reimagining the way people come together. https://corp.roblox.com