layout: single title: "Luau Type Checking Beta"

Hello!

We’ve been quietly working on building a type checker for Lua for quite some time now. It is now far enough along that we’d really like to hear what you think about it.

I am very happy to offer a beta test into the second half of the Luau effort.

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-type-checking-beta/).]

Beta Test

First, a word of caution: In this test, we are changing the syntax of Lua. We are pretty sure that we’ve mostly gotten things right, but part of the reason we’re calling this a beta is that, if we learn that we’ve made a mistake, we’re going to go back and fix it even if it breaks compatibility.

Please try it out and tell us what you think, but be aware that this is not necessarily our final form.

Beta testers can try it out by enabling the “Enable New Lua Script Analysis” beta feature in Roblox Studio.

Overview

Luau is an ahead-of-time typechecking system that sits atop ordinary Lua code. It does not (yet) feed into the runtime system; it behaves like a super powerful lint tool to help you find bugs in your code quickly.

It is also what we call a gradual type system. This means that you can choose to add type annotations in some parts of your code but not others.

Two Modes

Luau runs in one of two modes: strict, and nonstrict.

Nonstrict Mode

Nonstrict mode is intended to be as helpful as possible for programs that are written without type annotations. We want to report whatever we can without reporting an error in reasonable Lua code.

- If a local variable does not have a type annotation and it is not initially assigned a table, its type is any
- Unannotated function parameters have type any

- We do not check the number of values returned by a function
- Passing too few or too many arguments to a function is ok

Strict Mode

Strict mode is expected to be more useful for more complex programs, but as a side effect, programs may need a bit of adjustment to pass without any errors.

- The types of local variables, function parameters, and return types are deduced from how they are used
- Errors are produced if a function returns an inconsistent number of parameters, or if it is passed the wrong number of arguments

Strict mode is not enabled by default. To turn it on, you need to add a special comment to the top of your source file.

```
--!strict
```

New syntax

You can write type annotations in 5 places:

- After a local variable
- After a function parameter
- After a function declaration (to declare the function’s return type)
- In a type alias, and
- After an expression using the new as keyword.

```
local foo: number = 55

function is_empty(param: string) => boolean
    return 0 == param:len()
end

type Point = {x: number, y: number}

local baz = quux as number
```

Type syntax

Primitive types

`nil`, `number`, `string`, and `boolean`

any

The special type any signifies that Luau shouldn't try to track the type at all. You can do anything with an any.

Tables

Table types are surrounded by curly braces. Within the braces, you write a list of name: type pairs:

```
type Point = {x: number, y: number}
```

Table types can also have indexers. This is how you describe a table that is used like a hash table or an array.

```
type StringArray = {[number]: string}


type StringNumberMap = {[string]: number}
```

Functions

Function types use a => to separate the argument types from the return types.

```
type Callback = (string) => number
```

If a function returns more than one value, put parens around them all.

```
type MyFunction = (string) => (boolean, number)
```

Unions

You can use a `|` symbol to indicate an "or" combination between two types. Use this when a value can have different types as the program runs.

```
function ordinals(limit)
    local i = 0
    return function() => number | nil
        if i < limit then
            local t = i
            i = i + 1
            return t
        else
            return nil
        end
    end
end
```

Options

It’s pretty commonplace to have optional data, so there is extra syntax for describing a union between a type and `nil`. Just put a `?` on the end. Function arguments that can be `nil` are understood to be optional.

```
function foo(x: number, y: string?) end

foo(5, 'five') -- ok
foo(5) -- ok
foo(5, 4) -- not ok
```

Type Inference

If you don’t write a type annotation, Luau will try to figure out what it is.

```
--!strict
local Counter = {count=0}

function Counter:incr()
    self.count = 1
    return self.count
end

print(Counter:incr()) -- ok
print(Counter.incr()) -- Error!
print(Counter.amount) -- Error!
```

Future Plans

This is just the first step!

We’re excited about a whole bunch of stuff:

- Nonstrict mode is way more permissive than we’d like
- Generics!
- Editor integration

layout: single title: "Luau Recap: February 2020"

We continue to iterate on our language stack, working on many features for type checking, performance, and quality of life. Some of them come with announcements, some come with release notes, and some just ship - here we will talk about all things that happened since November last year.

A lot of people work on these improvements; thanks @Apakovtac, @EthicalRobot, @fun_enthusiast, @xyzzyismagic, @zeuxcg!

[Originally posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-february-2020/).]

We were originally intending to ship the beta last year but had to delay it due to last minute bugs. However, it’s now live as a beta option on production! Go here to learn more:

EDIT: Please DO NOT publish places with type annotations just yet as they will not work on production! This is why it’s a beta However, please continue to experiment in Studio and give us feedback. We are reading everything and will be fixing reported bugs and discussing syntax / semantics issues some people brought up. Hello! We’ve been quietly working on building a type checker for Lua for quite some time now. It is now far enough along that we’d really like to hear what…

We’re continuing to iterate on the feedback we have received here. Something that will happen next is that we will enable type annotations on live server/clients - which will mean that you will be able to publish source code with type annotations without breaking your games. We still have work to do on the non-strict and strict mode type checking before the feature can move out of beta though, in particular we’ve implemented support for require statement and that should ship next week

We also fixed a few bugs in the type definitions for built-in functions/API and the type checker itself:

- `table.concat` was accidentally treating the arguments as required
- `string.byte` and `string.find` now have a correct precise type
- `typeof` comparisons in if condition incorrectly propagated the inferred type into `elseif` branches

We are also making the type checker more ergonomic and more correct. Two changes I want to call out are:

- Type aliases declared with `type X = Y` are now co-recursive, meaning that they can refer to each other, e.g.

```
type array<T> = { [number]: T }


type Wheel = { radius: number, car: Car }
type Car = { wheels: array<Wheel> }
```

- We now support type intersections `(A & B)` in addition to type unions `(A | B)`. Intersections are critical to modeling overloaded functions correctly - while Lua as a language doesn't support function overloads, we have various APIs that have complex overloaded semantics - one of them that people who tried the beta had problems with was UDim2.new - and it turns out that to correctly specify the type, we had to add support for intersections. This isn't really intended as a feature that is used often in scripts developers write, but it's important for internal use.

Debugger (beta)

When we shipped the original version of the VM last year, we didn't have the debugger fully working. Debugger relies on low-level implementation of the old VM that we decided to remove from the new VM - as such we had to make a new low-level debugging engine.

This is now live under the Luau VM beta feature, see this post (https://devforum.roblox.com/t/luau-in-studio-beta/456529) for details.

If you use the debugger at all, please enable the beta feature and try it out - we want to fix all the bugs we can find, and this is blocking us enabling the new VM everywhere.

(a quick aside: today the new VM is enabled on all servers and all clients, and it's enabled in Studio "edit" mode for plugins - but not in Studio play modes, and full debugger support is what prevents us from doing so)

Language

This section is short and sweet this time around:

- You can now use continue statement in for/while/repeat loops. :tada:

Please note that we only support this in the new VM, so you have to be enrolled in Luau VM beta to be able to use it in Studio. It will work in game regardless of the beta setting as per above.

Performance

While we have some really forward looking ideas around multi-threading and native code compilation that we're starting to explore, we also continue to improve performance across the board based on our existing performance backlog and your feedback.

In particular, there are several memory and performance optimizations that shipped in the last few months:

- Checking for truth (`if foo or foo and bar`) is now a bit faster, giving 2-3% performance improvements on some benchmarks
- `table.create` (with value argument) and `table.pack` have been reimplemented and are ~1.5x faster than before

- Internal mechanism for filling arrays has been made faster as well, which makes `Terrain:ReadVoxels` ~10% faster
- Catching engine-generated errors with pcall/xpcall is now ~1.5x faster (this only affects performance of calls that generated errors)
- String objects now take 8 bytes less memory per object (and in an upcoming change we'll save a further 4 bytes)
- Capturing local variables that are never assigned to in closures is now much faster, takes much less memory and generates much less GC pressure. This can make closure creation up to 2x faster, and improves some Roact benchmarks by 10%. This is live in Studio and will ship everywhere else shortly.
- The performance of various for loops (numeric & ipairs) on Windows regressed after a VS2017 upgrade; this regression has been fixed, making all types of loops perform roughly equally. VS2017 upgrade also improved Luau performance on Windows by ~10% across the board.
- Lua function calls have been optimized a bit more, gaining an extra 10% of performance in call-heavy benchmarks on Windows.
- Variadic table constructors weren't compiled very efficiently, resulting in surprisingly low performance of constructs like `{...}`. Fixing that made `{...}` ~3x faster for a typical number of variadic arguments.

Diagnostics

We spent some time to improve error messages in various layers of the stack based on the reports from community. Specifically:

- The static analysis warning about incorrect bounds for numeric for loops is now putting squigglies in the right place.
- Fixed false positive static analysis warnings about unreachable code inside repeat…until loops in certain cases.
- Multiline table construction expressions have a more precise line information now which helps in debugging since callstacks are now easier to understand
- Incomplete statements (e.g. foo) now produce a more easily understandable parsing error
- In some cases when calling the method with a **.** instead of **:**, we emitted a confusing error message at runtime (e.g. humanoid.LoadAnimation(animation)). We now properly emit the error message asking the user if **:** was intended.
- The legacy global `ypcall` is now flagged as deprecated by script analysis
- If you use a Unicode symbol in your source program outside of comments or string literals, we now produce a much more clear message, for example:

```
local pi = 3.13 -- spoiler alert: this is not a dot!
```

produces `Unexpected Unicode character: U+2024. Did you mean '.'?`

LoadLibrary removal

Last but not least, let's all press F for LoadLibrary (https://devforum.roblox.com/t/loadlibrary-is-going-to-be-removed-on-february-3rd/382516).

It was fun while it lasted, but supporting it caused us a lot of pain over the years and prevented some forward-looking changes to the VM. We don’t like removing APIs from the platform, but in this case it was necessary. Thanks to the passionate feedback from the community we adjusted our initial rollout plans to be less aggressive and batch-processed a lot of gear items that used this function to stop using this function. The update is in effect and LoadLibrary is no more.

As usual, if you have any feedback about any of these updates, suggestions, bug reports, etc., post them in this thread or (preferably for bugs) as separate posts in the bug report category.

layout: single title: "Luau Recap: May 2020"

Luau (lowercase u, “l-wow”) is an umbrella initiative to improve our language stack - the syntax, compiler, virtual machine, builtin Lua libraries, type checker, linter (known as Script Analysis in Studio), and more related components. We continuously develop the language and runtime to improve performance, robustness and quality of life. Here we will talk about all things that happened since the update in March!

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-may-2020/).]

New function type annotation syntax

As noted in the previous update, the function type annotation syntax now uses `:` on function definitions and `->` on standalone function types:

```
type FooFunction = (number, number) -> number


function foo(a: number, b: number): number
    return a + b
end
```

This was done to make our syntax more consistent with other modern languages, and is easier to read in type context compared to our old `=>`.

This change is now live; the old syntax is still accepted but it will start producing warnings at some point and will be removed eventually.

Number of locals in each function is now limited to 200

As detailed in Upcoming change to (correctly) limit the local count to 200 (https://devforum.roblox.com/t/upcoming-change-to-correctly-limit-the-local-count-to-200/528417) (which is now live), when we first shipped Luau we accidentally set the local limit to 255 instead of 200. This resulted in confusing error messages and code that was using close to 250 locals was very fragile as it could easily break due to minor codegen changes in our compiler.

This was fixed, and now we’re correctly applying limits of 200 locals, 200 upvalues and 255 registers (per function) - and emit proper error messages pointing to the right place in the code when either limit is exceeded.

This is technically a breaking change but scripts with >200 locals didn't work in our old VM and we felt like we had to make this change to ensure long-term stability.

Require handling improvements in type checker + export type

We're continuing to flesh out the type checker support for modules. As part of this, we overhauled the require path tracing - type checker is now much better at correctly recognizing (statically) which module you're trying to require, including support for `game:GetService`.

Additionally, up until now we have been automatically exporting all type aliases declared in the module (via `type X = Y`); requiring the module via `local Foo = require(path)` made these types available under `Foo.` namespace.

This is different from the explicit handling of module entries, that must be added to the table returned from the `ModuleScript`. This was highlighted as a concern, and to fix this we've introduced `export type` syntax.

Now the only types that are available after require are types that are declared with `export type X = Y`. If you declare a type without exporting it, it's available inside the module, but the type alias can't be used outside of the module. That allows to cleanly separate the public API (types and functions exposed through the module interface) from implementation details (local functions etc.).

Improve type checker robustness

As we're moving closer to enabling type checking for everyone to use (no ETA at the moment), we're making sure that the type checker is as robust as possible.

This includes never crashing and always computing the type information in a reasonable time frame, even on obscure scripts like this one:

```
type ( ... ) ( ) ;
( ... ) ( - - ... ) ( - ... )
type = ( ... ) ;
( ... ) (  ) ( ...  )  ;
( ... ) ""
```

To that end we've implemented a few changes, most of them being live, that fix crashes and unbounded recursion/iteration issues. This work is ongoing, as we're fixing issues we encounter in the testing process.

Better types for Lua and Roblox builtin APIs

In addition to improving the internals of the type checker, we're still working on making sure that the builtin APIs have correct type information exposed to the type checker.

In the last few weeks we've done a major audit and overhaul of that type information. We used to have many builtin methods "stubbed" to have a very generic type like `any` or `(...) -> any`, and while we still have a few omissions we're much closer to full type coverage.

One notable exception here is the `coroutine.` library which we didn't get to fully covering, so the types for many of the functions there are imprecise.

If you find cases where builtin Roblox APIs have omitted or imprecise type information, please let us know by commenting on this thread or filing a bug report.

The full set of types we expose as of today is listed here for inquisitive minds: https://gist.github.com/zeux/d169c1416c0c65bb88d3a3248582cd13 (https://gist.github.com/zeux/d169c1416c0c65bb88d3a3248582cd13)

Removal of __gc from the VM

A bug with `continue` and local variables was reported to us a few weeks ago; the bug was initially believed to be benign but it was possible to turn this bug into a security vulnerability by getting access to `__gc` implementation for builtin Roblox objects. After fixing the bug itself (the turnaround time on the bug fix was about 20 hours from the bug report), we decided to make sure that future bugs like this don't compromise the security of the VM by removing `__gc`.

`__gc` is a metamethod that Lua 5.1 supports on userdata, and future versions of Lua extend to all tables; it runs when the object is ready to be garbage collected, and the primary use of that is to let the userdata objects implemented in C to do memory cleanup. This mechanism has several problems:

- `__gc` is invoked by the garbage collector without context of the original thread. Because of how our sandboxing works this means that this code runs at highest permission level, which is why `__gc` for newproxy-created userdata was disabled in Roblox a long time ago (10 years?)
- `__gc` for builtin userdata objects puts the object into non-determinate state; due to how Lua handles `__gc` in weak keys (see https://www.lua.org/manual/5.2/manual.html#2.5.2 (https://www.lua.org/manual/5.2/manual.html#2.5.2)), these objects can be observed by external code. This has caused crashes in some Roblox code in the past; we changed this behavior at some point last year.
- Because `__gc` for builtin objects puts the object into non-determinate state, calling it on the same object again, or calling any other methods on the object can result in crashes or vulnerabilities where the attacker gains access to arbitrarily mutating the process memory from a Lua script. We normally don't expose `__gc` because the metatables of builtin objects are locked but if it accidentally gets exposed the results are pretty catastrophic.
- Because `__gc` can result in object resurrection (if a custom Lua method adds the object back to the reachable set), during garbage collection the collector has to traverse the set of userdatas twice - once, to run `__gc` and a second time to mark the survivors.

For all these reasons, we decided that the `__gc` mechanism just doesn't pull its weight, and completely removed it from the VM - builtin userdata objects don't use it for memory reclamation anymore, and naturally declaring `__gc` on custom userdata objects still does nothing.

Aside from making sure we're protected against these kinds of vulnerabilities in the future, this makes garbage collection ~25% faster.

Memory and performance improvements

It's probably not a surprise at this point but we're never fully satisfied with the level of performance we get. From a language implementation point of view, any performance improvements we can make without changing the semantics are great, since they automatically result in Lua code running faster. To that end, here's a few changes we've implemented recently:

- ~A few string. methods, notably string.byte and string.char, were optimized to make it easier to write performant deserialization code. string.byte is now 4x faster than before for small numbers of returned characters. For optimization to be effective, it's important to call the function directly (`string.byte(foo, 5)`) instead of using method calls (`foo:byte(5)`). This had to be disabled due to a rare bug in some cases, this optimization will come back in a couple of weeks.
- `table.unpack` was carefully tuned for a few common cases, making it ~15% faster; `unpack` and `table.unpack` now share implementations (and the function objects are equal to each other).
- While we already had a very efficient parser, one long standing bottleneck in identifier parsing was fixed, making script compilation ~5% faster across the board, which can slightly benefit server startup times.
- Some builtin APIs that use floating point numbers as arguments, such as various `Vector3` constructors and operators, are now a tiny bit faster.
- All string objects are now 8 bytes smaller on 64-bit platforms, which isn't a huge deal but can save a few megabytes of Lua heap in some games.
- Debug information is using a special compact format that results in ~3.2x smaller line tables, which ends up making function bytecode up to ~1.5x smaller overall. This can be important for games with a lot of scripts.
- Garbage collector heap size accounting was cleaned up and made more accurate, which in some cases makes Lua heap ~10% smaller; the gains highly depend on the workload.

Library changes

The standard library doesn't see a lot of changes at this point, but we did have a couple of small fixes here:

- `coroutine.wrap` and `coroutine.create` now support C functions. This was the only API that treated Lua and C functions differently, and now it doesn't.
- `require` silently skipped errors in module scripts that occurred after the module scripts yielding at least once; this was a regression from earlier work on yieldable pcall and has been fixed.

As usual, if you have questions, comments, or any other feedback on these changes, feel free to share it in this thread or create separate posts for bug reports.

---

layout: single title: "Luau Recap: June 2020"

… otherwise known as “This Month in Luau” I guess? You know the drill by now. We’ll talk about exciting things that happened to Luau - our new language stack.

anxiously glances at FIB3 thread that casts a huge shadow on this announcement, but hopefully somebody will read this

Many people work on these improvements; thanks @Apakovtac, @EthicalRobot, @fun_enthusiast, @zeuxcg!

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-june-2020/).]

We have a website!

Many developers told us on many occasions that as much as they love the recaps, it’s hard to know what the state of the language or libraries is if the only way to find out is to read through all updates. What’s the syntax extensions that Luau supports now? How do I use type checking? What’s the status of from Lua 5.x?

Well, you can find all of this out here now: https://roblox.github.io/luau/ (https://roblox.github.io/luau/)

Please let us know if this documentation can be improved - what are you missing, what could be improved. For now to maximize change velocity this documentation is separate from DevHub; it’s also meant as an external resource for people who don’t really use the language but are curious about the differences and novelties.

Also, `_VERSION` now returns "Luau" because we definitely aren’t using Lua 5.1 anymore.

Compound assignments

A long-standing feature request for Lua is compound assignments. Somehow Lua never got this feature, but Luau now implements `+=`, `-=`, `*=`, `/=`, `%=`, `^=` and `..=` operators. We decided to implement them because they are absolutely ubiquitous among most frequently used programming languages, both those with C descent and those with different lineage (Ruby, Python). They result in code that’s easier to read and harder to make mistakes in.

We do not implement `++` and `--`. These aren’t universally adopted, `--` conflicts with comment syntax and they are arguably not as intuitively obvious. We trust everyone to type a few extra characters for `+= 1` without too much trouble.

Two important semantical notes are that the expressions on the left hand side are only evaluated once, so for example `table[makeIndex()] += 1` only runs `makeIndex` once, and that compound assignments still call all the usual metamethod (`__add` et al, and `__index`/`__newindex`) when necessary - you don’t need to change any data structures to work with these.

There's no noticeable performance improvement from these operators (nor does using them carry a cost) - use them when they make sense for readability.

Nicer error messages

Good errors are critical to be able to use Luau easily. We've spent some time to improve the quality of error messages during parsing and runtime execution:

- In runtime type errors, we now often use the "Roblox" type name instead of plain userdata, e.g. `math.abs(v)` now says `number` expected, got `Vector3`
- When arguments are just missing, we now explicitly say that they are missing in libraries like math/table; the old message was slightly more confusing
- `string.format` in some cases produced error messages that confused missing arguments for incorrect types, which has been fixed
- When a builtin function such as `math.abs` fails, we now add the function name to the error message. This is something that used to happen in Lua, then we lost this in Luau because Luau removes a very fragile mechanism that supported that, but we now have a new, robust way to report this so you can have the function name back! The message looks like this now: `invalid argument #1 to 'abs' (number expected, got nil)`
- In compile-time type errors, we now can identify the case when the field was mistyped with a wrong case (ha), and tell you to use the correct case instead.
- When you forget an `end` statement, we now try to be more helpful and point you to the problematic statement instead of telling you that the end is missing at the very end of the program. This one is using indentation as a heuristic so it doesn't always work perfectly.
- We now have slightly more helpful messages for cases when you forget parentheses after a function call
- We now have slightly more helpful messages for some cases when you accidentally use `( ... )` instead of `{ ... }` to create a table literal Additionally two places had very lax error checking that made the code more fragile, and we fixed those:
- `xpcall` now fails immediately when the error function argument is not a function; it used to work up until you get an error, and failed at that point, which made it hard to find these bugs
- `tostring` now enforces the return type of the result to be a string - previously `__tostring` could return a non-string result, which worked fine up until you tried to do something like passing the resulting value to `string.format` for `%s`. Now `tostring` will fail early. Our next focus here is better error messages during type checking - please let us know if there are other errors you find confusing and we could improve!

Type checker improvements

We're getting closer and closer to be able to move out of beta. A big focus this month was on fixing all critical bugs in the type checker - it now should never hang or crash Studio during type checking, which took a bit of work to iron out all the problems.

Notably, typing function string.length no longer crashes Studio (although why you'd do that is unclear), and Very Large Scripts With Tons Of Nested Statements And Expressions should be stable as well.

We've also cleaned up the type information for builtin libraries to make it even more precise, including a few small fixes to `string/math` functions, and a much more precise coroutine library type information. For the latter we've introduced a primitive type `thread`, which is what `coroutine` library works with.

Linter improvements

Linter is the component that produces warnings about scripts; it's otherwise known as "Static Analysis" in Studio, although that is now serving as a place where we show type errors as well.

Most of the changes here this month are internal as they concern warnings that aren't yet enabled in Studio (the web site linked above documents all warnings including ones that aren't active yet but may become active), but once notable feature is that you can now opt out of individual warnings on a script-by-script basis by adding a --!nolint comment to the top of the script. For example, if you really REALLY *REALLY* like the `Game` global, you can add this to the top of the script:

```
--!nolint DeprecatedGlobal
```

Or, if you basically just want us to not issue any warnings ever, I guess you can add this:

```
--!nocheck
--!nolint
```

and live happily ignorant of all possible errors up until you run your code. (please don't do that)

os. enhancements

Our overall goal is to try to be reasonably compatible with Lua 5.x in terms of library functions we expose. This doesn't always work - in some cases we have to remove library features for sandboxing reasons, and in others the library functions don't make sense in context of Roblox. However, some of these decisions can be revised later. In particular, when we re-added `os.` library to Roblox, we limited it to `os.date`, `os.time` and `os.difftime` (although why `difftime` is a thing isn't clear), omitting `os.clock` and restricting inputs to `os.date` to return a table with date components, whereas Lua 5.x supports format strings.

Well, this changes today. `os.clock` is now available if you need a high-precision time for benchmarking, and `os.date` can now return formatted date using Lua 5.x format string that you can read about here https://www.lua.org/pil/22.1.html (https://www.lua.org/pil/22.1.html) (we support all these specifiers: aAbBcdHIjmMpSUwWxXyYzZ).

While `os.date()` is hopefully welcome, `os.clock` may raise some eyebrows - aren't there enough timing functions in Roblox already? Well, this is nice if you are trying to port code from Lua 5.x to Luau, and there's this

![Oblig. xkcd]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-june-2020-xkcd.png)

But really, most existing Roblox timing functions are… problematic.

- `time()` returns the total amount of time the game has been running simulation for, it's monotonic and has reasonable precision. It's fine - you can use it to update internal gameplay systems without too much trouble. It should've been called "tick" perhaps but that ship has sailed.
- `elapsedTime` and its close cousin `ElapsedTime`, are telling you "how much time has elapsed since the current instance of Roblox was started.". While technically true, this isn't actually useful because on mobile the "start" time here can be days in the past. It's also inadequate for performance measurements as on Windows, it has a 1ms resolution which isn't really enough for anything interesting. We're going to deprecate this in the future.
- `tick()` sounds perfect - it has a high resolution (usually around 1 microsecond), and a well-defined baseline - it counts since UNIX epoch! Or, well, it actually doesn't. On Windows, it returns you a variant of the UNIX timestamp in local time zone. In addition, it can be off by 1 second from the actual, real UNIX timestamp, and might have other idiosyncrasies on non-Windows platforms. We're going to deprecate this in the future

So, if you need a UNIX timestamp, you should use `os.time()`. You get a stable baseline (from 1970's) and 1s resolution. If you need to measure performance, you should use `os.clock()`. You don't get a stable baseline, but you get ~1us resolution. If you need to do anything else, you should probably use `time()`.

Performance optimizations

As you can never have too much performance, we're continuing to work on performance! We're starting to look into making Vector3 faster and improving the garbage collector, with some small changes already shipping, but overall it's a long way out so here are the things that did get visibly better:

- A few `string.` methods, notably `string.byte` and `string.char`, were optimized to make it easier to write performant deserialization code. string.byte is now ~4x faster than before for small numbers of returned characters. For optimization to be effective, it's important to call the function directly (`string.byte(foo, 5)`) instead of using method calls (`foo:byte(5)`)
- Optimize coroutine resumption, making some code that is heavily reliant on `coroutine`. library ~10% faster. We have plans to improve this further, watch this space.
- Optimize `typeof()` to run ~6x faster. It used to be that `type()` was much faster than `typeof()` but they now should be more or less comparable.
- Some secret internal optimizations make some scripts a few percent faster
- The memory allocator used in Luau was rewritten using a new, more efficient, implementation. There might be more changes here in the future to save some memory, but for now this makes some allocation-intensive benchmarks ~15% faster.
- Using tables with keys that are not strings or numbers is a fair bit more efficient now (most commonly comes up when Instance is used as a key in a hash table), on par with using strings.

Also we found a bug with some of our optimizations (which delayed the string. performance improvement above, but also could affect some math. calls) where in some complex functions you would see valid calls to math. etc. breaking with non-sensical errors such as “expected number, got table” - this has been fixed!

Memory optimizations

As with performance, our goal here is simple - the more efficient internal Luau structures can become, the less memory will Lua heap take. This is great for both memory consumption, and for garbage collection performance as the collector needs to traverse less data. There’s a few exciting changes in this area this month:

- Non-array-like tables now take 20% less space. This doesn’t affect arrays but can be observed on object-like tables, both big and small. This is great because some of you are using a lot of large tables apparently, since this resulted in very visible reduction in overall Lua heap sizes across all games.
- Function objects now take up to 30% less space. This isn’t as impactful since typically function objects are not created very frequently and/or don’t live for very long, but it’s nice nonetheless.
- New allocator mentioned in the previous section can save up to 5-6% of Lua heap memory as well, although these gains are highly dependent on the workload, and we usually see savings in the 1-2% range.

And that’s it! Till next time. As usual let us know if you have questions, suggestions or bug reports.

layout: single title: "Luau Recap August 2020"

As everyone knows by now, Luau is our new language stack that you can read more about at [https://roblox.github.io/luau (https://roblox.github.io/luau)](https://roblox.github.io/luau) and the month following June is August so let’s talk about changes, big and small, that happened since June!

Many people work on these improvements, with the team slowly growing - thanks @Apakovtac, @EthicalRobot, @fun_enthusiast, @mrow_pizza and @zeuxcg!

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-august-2020/).]

Type annotations are safe to use in production!

When we started the Luau type checking beta, we’ve had a big warning sign in the post saying to not publish the type-annotated scripts to your production games which some of you did anyway. This was because we didn’t want to commit to specific syntax for types, and were afraid that changing the syntax would break your games.

This restriction is lifted now. All scripts with type annotations that parse & execute will continue to parse & execute forever. Crucially, for this to be true you must not be using old fat arrow syntax for functions, which we warned you about for about a month now:

![Fat arrow deprecated]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-august-2020-arrow.png)

… and must not be using the `__meta` property which no longer holds special meaning and we now warn you about that:

![meta deprecated]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-august-2020-meta.png)

Part of the syntax finalization also involved changing the precedence on some type annotations and adding support for parentheses; notably, you can now mix unions and intersections if you know what that means (`(A & B) | C` is valid type syntax). Some complex type annotations changed their structure because of this - previously `(number) -> string & (string) -> string` was a correct way to declare an intersection of two function types, but now to keep it parsing the same way you need to put each function type in parentheses: `((number) -> string) & ((string) -> string)`.

Type checking is not out of beta yet - we still have some work to do on the type checker itself. The items on our list before going out of beta right now include:

- Better type checking for unary/binary operators
- Improving error messages to make type errors more clear

- Fixing a few remaining crashes for complex scripts
- Fixing conflation of warnings/errors between different scripts with the same path in the tree
- Improving type checking of globals in nonstrict mode (strict mode will continue to frown upon globals)

Of course this doesn't mark the end of work on the feature - after type checking goes out of beta we plan to continue working on both syntax and semantics, but that list currently represents the work we believe we have left to do in the first phase - please let us know if there are other significant issues you are seeing with beta beyond future feature requests!

Format string analysis

A few standard functions in Luau are using format strings to dictate the behavior of the code. There's `string.format` for building strings, `string.gmatch` for pattern matching, `string.gsub`'s replacement string, `string.pack` binary format specification and `os.date` date formatting.

In all of these cases, it's important to get the format strings right - typos in the format string can result in unpredictable behavior at runtime including errors. To help with that, we now have a new lint rule that parses the format strings and validates them according to the expected format.

![String format]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-august-2020-format.png)

Right now this support is limited to direct library calls (`string.format("%.2f", ...)` and literal strings used in these calls - we may lift some of these limitations later to include e.g. support for constant locals.

Additionally, if you have type checking beta enabled, string.format will now validate the argument types according to the format string to help you get your `%d`s and `%s`es right.

![String format]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-august-2020-format2.png)

Improvements to string. library

We've upgraded the Luau string library to follow Lua 5.3 implementation; specifically:

- `string.pack/string.packsize/string.unpack` are available for your byte packing needs
- `string.gmatch` and other pattern matching functions now support `%g` and `\0` in patterns

This change also [inadvertently] makes `string.gsub` validation rules for replacement string stricter - previously `%` followed by a non-digit character was silently accepted in a replacement string, but now it generates an error. This accidentally broke our own localization script Purchase Prompt broken in some games (% character in title) (https://devforum.roblox.com/t/purchase-prompt-broken-in-some-games-character-in-title/686237)), but we got no other reports, and this in retrospect is a good change as it makes future extensions to string replacement safe… It was impossible for us to roll the change back and due to a long release window because of an internal company holiday we decided to keep the change as is, although we'll try to be more careful in the future.

On a happier note, string.pack may seem daunting but is pretty easy to use to pack binary data to reduce your network traffic (note that binary strings aren't safe to use in DataStores currently); I've posted an example in the release notes thread Release Notes for 441 (https://devforum.roblox.com/t/release-notes-for-441/686773) that allows you to pack a simple character state in 16 bytes like this:

```
local characterStateFormat = "fffbbbB"

local characterState = string.pack(characterStateFormat,
    posx, posy, posz, dirx * 127, diry * 127, dirz * 127, health)
```

And unpack it like this after network transmission:

```
local posx, posy, posz, dirx, diry, dirz, health =
    string.unpack(characterStateFormat, characterState)
dirx /= 127
diry /= 127
dirz /= 127
```

Assorted fixes

As usual we fixed a few small problems discovered through testing. We now have an automated process that generates random Luau code in semi-intelligent ways to try to break different parts of our system, and a few fixes this time are a direct result of that.

- Fix line debug information for multi-line function calls to make sure errors for code like `foo.Bar(...)` are generated in the appropriate location when foo is nil
- Fix debug information for constant upvalues; this fixes some bugs with watching local variables from the nested functions during debugging
- Fix an off-by-one range check in string.find for init argument that could result in reading uninitialized memory
- Fix type confusion for table.move target table argument that could result in reading or writing arbitrary memory
- Fix type confusion for `debug.getinfo` in some circumstances (we don't currently expose getinfo but have plans to do so in the future)
- Improve out of memory behavior for large string allocations in string.rep and some other functions like `table.concat` to handle these conditions more gracefully
- Fix a regression with `os.time` from last update, where it erroneously reverted to Lua 5.x behavior of treating the time as a local time. Luau version (intentionally) deviates from this by treating the input table as UTC, which matches `os.time()` behavior with no arguments.

Performance improvements

Only two changes in this category here this time around; some larger scale performance / memory improvements are still pending implementation.

- Constant locals are now completely eliminated in cases when debugging is not available (so on server/client), making some scripts ~1-2% faster
- Make script compilation ~5% faster by tuning the compiler analysis and code generation more carefully Oh, also `math.round` is now a thing which didn’t fit into any category above.

layout: single title: "Luau Recap: October 2020"

Luau is our new language that you can read more about at https://roblox.github.io/luau (https://roblox.github.io/luau); we've been so busy working on the current projects that we didn't do an update in September, so let's look at changes that happened since August!

Many people work on these improvements, with the team slowly growing - thanks @Apakovtac, @EthicalRobot, @fun_enthusiast, @machinamentum, @mrow_pizza and @zeuxcg!

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-october-2020/).]

Types are very close

We've been in beta for a while now, but we're steadily marching towards getting the first release of the type checker, what we call "types v0", out of the door. It turns out that we've substantially underestimated the effort required to make the type system robust, strike the balance between "correct" and "usable" and give quality diagnostics in the event we do find issues with your code □

Because of this, we're changing the original plans for the release a bit. We're actively working on a host of changes that we consider to be part of the "v0" effort, and when they are all finished - which should happen next month, fingers crossed - we're going to be out of beta!

However, by default, on scripts with no annotations, we won't actually activate type checking. You would have to opt into the type checking by using `--!nonstrict` or `--!strict`, at the top of each script. We are also going to open the second beta, "All scripts use non-strict mode by default" or something along these lines.

This is important because we found that our non-strict mode still needs some more work to be more tolerant to some code that occurs commonly in Roblox and is correct, but doesn't type-check. We're going to evaluate what changes specifically are required to make this happen, but we didn't want the extra risk of a flood of reports about issues reported in existing code to shift the release date in an unpredictable fashion.

To that end, we've been working on Lots and Lots and Lots and Lots and Lots of changes to finish the first stage. Some of these changes are already live and some are rolling out; the amount of changes is so large that I can't possibly list the up-to-date status on each one as these recaps are synthesized by the human who is writing this on a Friday night, so here's just a raw list of changes that may or may not have been enabled:

- Strict mode is now picky about passing extra arguments to functions, even though they are discarded silently at runtime, as this can hide bugs
- The error message about using a : vs . during type checking is now much more precise
- Recursive type annotations shouldn't crash the type checker now, and we limit the recursion and iteration depth during type checking in a few cases in general in an effort to make sure type checker always returns in finite time

- Binary relational operators (< et al) are now stricter about the argument types and infer the argument types better
- Function argument and return types are now correctly contra- and co-variant; if this looks like gibberish to you, trust me - it's for the best!
- Fixed a few problems with indexing unions of tables with matching key types
- Fixed issues with tracing types across modules (via require) in non-strict mode
- Error messages for long table types are now trimmed to make the output look nicer
- Improve the interaction between table types of unknown shape (`{ [string]: X }`) and table types of known shape.
- Fix some issues with type checking table assignments
- Fix some issues with variance of table fields
- Improve the legibility of type errors during function calls - errors now point at specific arguments that are incorrect, and mismatch in argument count should clearly highlight the problem
- Fix types for many builtins including `ipairs`, `table.create`, `Color3.fromHSV`, and a few others
- Fix missing callbacks for some instance types like `OnInvoke` for bindables (I think this one is currently disabled while we're fixing a semi-related bug, but should be enabled soon!)
- Rework the rules under which globals are okay to use in non-strict mode to mostly permit valid scripts to type-check; strict mode will continue to frown upon the use of global variables
- Fix a problem with the beta where two scripts with identical names would share the set of errors/warnings, resulting in confusing error highlights for code that doesn't exist
- Improve the legibility of type errors when indexing a table without a given key
- Improve the parsing error when trying to return a tuple; `function f(): string, number` is invalid since the type list should be parenthesized because of how our type grammar is currently structured
- Type checker now clearly reports cases where it finds a cyclic dependency between two modules
- Type errors should now be correctly sorted in the Script Analysis widget
- Error messages on mismatches between numbers of values in return statements should now be cleaner, as well as the associated type mismatch errors
- Improve error messages for comparison operators
- Flag attempts to require a non-module script during type checking
- Fix some cases where a type/typeof guard could be misled into inferring a non-sensible type
- Increase the strictness of return type checks in strict mode - functions now must conform to the specified type signature, whereas before we'd allow a function to return no values even in strict mode
- Improve the duplicate definition errors to specify the line of the first definition
- Increase the strictness of binary operators in strict mode to enforce the presence of the given operator as a built-in or as part of the metatable, to make sure that strict mode doesn't infer types when it can't guarantee correctness
- Improve the type errors for cyclic types to make them more readable
- Make type checker more friendly by rewording a lot of error messages
- Fix a few crashes in the type checker (although a couple more remain - working on them!)
- … I think that's it?
- …edit ah, of course I forgot one thing - different enums that are part of the Roblox API now have distinct types and you can refer to the types by name e.g. `Enum.Material`; this should go live next week though. If you want to pretend that you've read and understood the entire list above, just know that we've worked on making sure strict mode is more reliably reporting type errors and doesn't infer types incorrectly, on making sure non-strict

mode is more forgiving for code that is probably valid, and on making the type errors more specific, easier to understand, and correct.

Type syntax changes

There's only two small changes here this time around - the type syntax is now completely stable at this point, and any existing type annotation will continue parsing indefinitely. We of course reserve the right to add new syntax that's backwards compatible :slight_smile:

On that note, one of the small changes is that we've finally removed support for fat arrows (`=>`); we've previously announced that this would happen and that thin arrows (`->`) are the future, and had warnings issued on the legacy syntax for a while. Now it's gone.

On a positive note, we've added a shorter syntax for array-like table types. Whereas before you had to use a longer `{ [number]: string }` syntax to declare an array-like table that holds strings, or had to define an `Array` type in every. single. module. you. ever. write. ever., now you can simply say `{string}`! This syntax is clean, in line with the value syntax for Lua table literals, and also was chosen by other research projects to add type annotations to Lua.

(if you're a monster that uses mixed tables, you'll have to continue using the longer syntax e.g. `{ [number]: string, n: number }`)

Library changes

There's only a few small tweaks here this time around on the functionality front:

- `utf8.charpattern` is now exactly equal to the version from Lua 5.3; this is now possible because we support `\0` in patterns, and was suggested by a user on devforum. We do listen!
- `string.pack` now errors out early when the format specifier is Way Too Large. This was reported on dev forum and subsequently fixed. Note that trying to generate a Moderately Large String (like, 100 MB instead of 100 GB) will still succeed but may take longer than we'd like - we have a plan to accelerate operations on large strings substantially in the coming months.

Performance improvements

We were super focused on other things so this is very short this time around. We have a lot of ideas here but they are waiting for us to finish some other large projects!

- Method calls on strings via `:` are now ~10% faster than before. We still recommend using fully-qualified calls from string library such as `string.foo(str)`, but extra performance never hurts!
- Speaking of string methods, string.sub is now ~20% faster than before with the help of voodoo magic.

Miscellaneous fixes

There were a few small fixes that didn't land into any specific category that I wanted to highlight:

- In some rare cases, debug information on conditions inside loops have been fixed to stop debugger from incorrectly suggesting that the current line is inside a branch that wasn't taken. As usual, if you ever see debugger misbehaving, please file bugs on this!
- Code following `assert(false)` is now treated as an unreachable destination from the linting and type checking point of view, similarly to error calls.
- Linting support for various format strings has been greatly improved based on fantastic feedback from @Halalaluyafail3 (thanks!).

Ok, phew, that's what I get for skipping a month again. Please don't hesitate to report bugs or suggestions, here or via separate posts. Due to our usual end-of-year code freeze there's going to be one more recap at the end of the year where we will look back at 2020 and take a small peek into the future.

---

layout: single title: "Luau Type Checking Release"

10 months ago, we've started upon the journey of helping Roblox scripters write robust code by introducing an early beta of type checking (https://devforum.roblox.com/t/luau-type-checking-release). We've received a lot of enthusiastic feedback and worked with the community on trying to make sure critical issues are addressed, usability is improved and the type system is ready for prime time.

Today I'm incredibly excited to announce that the first release of Luau (https://roblox.github.io/luau/) type checking is officially released! Thanks a lot to @Apakovtac, @EthicalRobot, @fun_enthusiast, @machinamentum, @mrow_pizza and @zeuxcg!

[Originally posted on the Roblox Developer Forum (https://devforum.roblox.com/t/luau-type-checking-release/).]

What is type checking?

When Luau code runs, every value has a certain type at runtime - a kind of value it stores. It could be a number, a string, a table, a Roblox Instance or one of many others. Thing is, some operations work on some types but don't work on others!

Consider this:

```
local p = Instance.new("Part")
p.Positio = Vector3.new(1,2,3)
```

Is this code correct? No - there's a typo. The way you get to find this typo is by running your code and eventually seeing an error message. Type checker tries to analyze your code before running, by assigning a type to each value based on what we know about how that value was produced, or based on the type you've explicitly told us using a new syntax extension, and can produce an error ahead of time:

!["Positio not found in class Part"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-type-checking-release-screenshot.png)

This can require some effort up front, especially if you use strict mode, but it can save you valuable time in the future. It can be especially valuable if you have a large complex code base you need to maintain for years, as is the case with many top Roblox games.

How do I use type checking?

A very important feature of Luau type checking you need to know about is that it has three modes:

- `nocheck`, where we don't type check the script in question.

- `nonstrict`, where we type check the script but try to be lenient to allow commonly seen patterns even if they may violate type safety
- `strict`, where we try to make sure that every single line of code you write is correct, and every value has a known type.

The modes can be selected per script by writing a comment at the top of the script that starts with `--!`, e.g. `--!strict`.

As of this release, the default mode is nocheck. This means by default you actually won't see the type checking produce feedback on your code! We had to use nocheck by default because we aren't fully ready to unleash nonstrict mode on unsuspecting users - we need to do a bit more work to make sure that most cases where we tell you that something is wrong are cases where yes, something is actually wrong.

However we highly encourage trying at least non-strict mode on your codebase. You can do this by opting into a different default via a Studio beta:

!["Studio option"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-type-checking-release-studio-option.png)

This beta only changes the default mode. Another way to change the mode is to prepend a `--!` comment to the script - you can do this manually for now, but if anyone in the community wants to release a plugin that does it automatically on selected scripts (+ descendants), that would be swell!

If you really want your code to be rock solid, we recommend trying out strict mode. Strict mode will require you to use type annotations.

What are type annotations and how do I use them?

Glad you asked! (please pretend you did) Type annotations are a way to tell the type checker what the type of a variable is. Consider this code in strict mode:

```
function add(x, y)
    return x + y
end
```

Is this code correct? Well, that depends. `add(2, 3)` will work just fine. `add(Vector3.new(1, 2, 3), Vector3.new(4, 5, 6))` will work as well. But `add({}, nil)` probably isn't a good idea.

In strict mode, we will insist that the type checker knows the type of all variables, and you'll need to help the type checker occasionally - by adding types after variable names separated by `:`:

```
function add(x: number, y: number)
    return x + y
end
```

If you want to tell the type checker “assume this value can be anything and I will take responsibility”, you can use `any` type which will permit any value of any type.

If you want to learn more about the type annotation syntax, you should read this documentation on syntax (https://roblox.github.io/luau/syntax.html#type-annotations). We also have a somewhat more complete guide to type checking than this post can provide, that goes into more details on table types, OOP, Roblox classes and enums, interaction with require and other topics - read it if you’re curious! (https://roblox.github.io/luau/typecheck.html).

What happens when I get a type error?

One concept that’s very important to understand is that right now type errors do not influence whether the code will run or not.

If you have a type error, this means that our type checker thinks your code has a bug, or doesn’t have enough information to prove the code works fine. But if you really want to forge ahead and run the code - you should feel free to do so!

This means that you can gradually convert your code to strict mode by adding type annotations and have the code runnable at all times even if it has type errors.

This also means that it’s safe to publish scripts even if type checker is not fully happy with them - type issues won’t affect script behavior on server/client, they are only displayed in Studio.

Do I have to re-learn Lua now?!?

This is a question we get often! The answer is “no”.

The way the type system is designed is that it’s completely optional, and you can use as many or as few types as you’d like in your code.

In non-strict mode, types are meant as a lightweight helper - if your code is likely wrong, we’re going to tell you about it, and it’s up to you on whether to fix the issue, or even disable the type checker on a given problematic file if you really don’t feel like dealing with this.

In strict mode, types are meant as a power user tool - they will require more time to develop your code, but they will give you a safety net, where changing code will be much less likely to trigger errors at runtime.

Is there a performance difference?

Right now type annotations are ignored by our bytecode compiler; this means that performance of the code you write doesn’t actually depend on whether you use strict, nonstrict or nocheck modes or if you have type annotations.

This is likely going to change! We have plans for using the type information to generate better bytecode in certain cases, and types are going to be instrumental to just-in-time compilation, something that we’re going to invest time into next year as well.

Today, however, there’s no difference - type information is completely elided when the bytecode is built, so there is zero runtime impact one way or another.

What is next for types?

This is the first full release of type checking, but it’s by far the last one. We have a lot more ground to cover. Here’s a few things that we’re excited about that will come next:

- Making nonstrict mode better to the point where we can enable it as a default for all Roblox scripts
- Adding several features to make strict mode more powerful/friendly, such as typed variadics, type ascription and better generics support
- Improving type refinements for type/typeof and nil checks
- Making it possible to view the type of a variable in Studio
- Reworking autocomplete to use type information instead of the current system

If you have any feedback on the type system, please don’t hesitate to share it here or in dedicated bug report threads. We’re always happy to fix corner cases that we’ve missed, fix stability issues if they are discovered, improve documentation when it’s not clear or improve error messages when they are hard to understand.

layout: single title: "Luau Recap: February 2021"

Luau is our new language that you can read more about at https://roblox.github.io/luau (https://roblox.github.io/luau). It's been a busy few months in Luau!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-february-2021/).]

Infallible parser

Traditional compilers have focused on tasks that can be performed on complete programs, such as type-checking, static analysis and code generation. This is all good, but most programs under development are incomplete! They may have holes, statements that will be filled in later, and lines that are in the middle of being edited. If we'd like to provide support for developers while they are writing code, we need to provide tools for incomplete programs as well as complete ones.

The first step in this is an *infallible* parser, that always returns an Abstract Syntax Tree, no matter what input it is given. If the program is syntactically incorrect, there will also be some syntax errors, but the parser keeps going and tries to recover from those errors, rather than just giving up.

The Luau parser now recovers from errors, which means, for example, we can give hints about programs in an IDE.

![A type error after a syntax error]({{ site.url }}{{ site.baseurl }}/assets/images/type-error-after-syntax-error.png)

Type assertions

The Luau type checker can't know everything about your code, and sometimes it will produce type errors even when you know the code is correct. For example, sometimes the type checker can't work out the intended types, and gives a message such as "Unknown type used... consider adding a type annotation".

!["Consider adding a type annotation"]({{ site.url }}{{ site.baseurl }}/assets/images/type-annotation-needed.png)

Previously the only way to add an annotation was to put it on the *declaration* of the variable, but now you can put it on the *use* too. A use of variable `x` at type `T` can be written `x :: T`. For example the type `any` can be used almost anywhere, so a common usage of type assertions is to switch off the type system by writing `x :: any`.

!["A type assertion y:any"]({{ site.url }}{{ site.baseurl }}/assets/images/type-annotation-provided.png)

Typechecking improvements

We've made various improvements to the Luau typechecker:

- We allow duplicate function definitions in non-strict mode.
- Better typechecking of `and`, `(f or g)()`, arrays with properties, and `string:format()`.
- Improved typechecking of infinite loops.
- Better error reporting for function type mismatch, type aliases and cyclic types.

Performance improvements

We are continuing to work on optimizing our VM and libraries to make sure idiomatic code keeps improving in performance. Most of these changes are motivated by our benchmark suite; while some improvements may seem small and insignificant, over time these compound and allow us to reach excellent performance.

- Table key assignments as well as global assignments have been optimized to play nicer with modern CPUs, yielding ~2% speedup in some benchmarks
- Luau function calls are now ~3% faster in most cases; we also have more call optimizations coming up next month!
- Modulo operation (%) is now a bit faster on Windows, resulting in ~2% performance improvement on some benchmarks

!["Benchmark vs Lua 5.3"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-february-2021-benchmark.png)

Debugger improvements

Our Luau VM implementation is focused on performance and provides a different API for implementation of debugger tools. But it does have its caveats and one of them was inability to debug coroutines (breakpoints/stepping).

The good news is that we have lifted that limitation and coroutines can now be debugged just like any regular function. This can especially help people who use Promise libraries that rely on coroutines internally.

![Debugging a coroutine]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-february-2021-debugger.png)

Library changes

`table` library now has a new method, `clear`, that removes all keys from the table but keeps the internal table capacity. When working with large arrays, this can be more efficient than assigning a table to `{}` - the performance gains are similar to that of using `table.create` instead of `{}` *when you expect the number of elements to stay more or less the same*. Note that large empty tables still take memory and are a bit slower for garbage collector to process, so use this with caution.

In addition to that we found a small bug in `string.char` implementation that allowed creating strings from out-of-range character codes (e.g. `string.char(2000)`); the problem has been fixed and these calls now correctly generate an error.

Coming soon...

- *Generic function types* will soon be allowed!

```
function id<a>(x: a): a
    return x
end
```

- *Typed variadics* will soon allow types to be given to functions with varying numbers of arguments!

```
function sum(...: number): number
    local result = 0
    for i,v in ipairs({...}) do
        result += v
    end
    return result
end
```

And there will be more!

---

layout: single title: "Luau Recap: March 2021"

Luau is our new language that you can read more about at https://roblox.github.io/luau (https://roblox.github.io/luau). It's been a busy month in Luau!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-march-2021/).]

Typed variadics

Luau supports *variadic* functions, meaning ones which can take a variable number of arguments (varargs!) but previously there was no way to specify their type. Now you can!

```
function f(x: string, ...: number)
  print(x)
  print(...)
end
f("hi")
f("lo", 5, 27)
```

This function takes a string, plus as many numbers as you like, but if you try calling it with anything else, you'll get a type error, for example `f("oh", true)` gives an error "Type `boolean` could not be converted into `number`"

Variadics can be used in function declarations, and function types, for example

```
type T = {
  sum: (...number) -> number
}
function f(x: T)
  print(x.sum(1, 2, 3))
end
```

Generic functions

WARNING Generic functions are currently disabled as we're fixing some critical bugs.

Typechecking improvements

We've made various improvements to the Luau typechecker:

- Check bodies of methods whose `self` has type `any`
- More precise types for `debug.*` methods
- Mutually dependent type aliases are now handled correctly

Performance improvements

We are continuing to squeeze the performance out of all sorts of possible code; this is an ongoing process and we have many improvements in the pipeline, big and small. These are the changes that are already live:

- Significantly optimized non-variadic function calls, improving performance by up to 10% on call-heavy benchmarks
- Improve performance of `math.clamp`, `math.sign` and `math.round` by 2.3x, 2x and 1.6x respectively
- Optimized `coroutine.resume` with ~10% gains on coroutine-heavy benchmarks
- Equality comparisons are now a bit faster when comparing to constants, including `nil`; this makes some benchmarks 2-3% faster
- Calls to builtin functions like `math.abs` or `bit32.rrotate` are now significantly faster in some cases, e.g. this makes SHA256 benchmark 25% faster
- `rawset`, `rawget`, `rawequal` and 2-argument `table.insert` are now 40-50% faster; notably, `table.insert(t, v)` is now faster than `t[#t+1]=v`

Note that we work off a set of benchmarks that we consider representative of the wide gamut of code that runs on Luau. If you have code that you think should be running faster, never hesitate to open a feature request / bug report on Roblox Developer Forum!

Debugger improvements

We continue to improve our Luau debugger and we have added a new feature to help with coroutine call debugging. The call stack that is being displayed while stopped inside a coroutine frame will display the chain of threads that have called it.

Before:

!["Old debugger"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-march-2021-debug-before.png)

After:

!["New debugger"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-march-2021-debug-after.png)

We have restored the ability to break on all errors inside the scripts. This is useful in cases where you need to track the location and state of an error that is triggered inside 'pcall'. For example, when the error that's triggered is not the one you expected.

!["Break on all exceptions"]({{ site.url }}{{ site.baseurl }}/assets/images/luau-recap-march-2021-debug-dialog.png)

Library changes

- Added the `debug.info` function which allows retrieving information about stack frames or functions; similarly to `debug.getinfo` from Lua, this accepts an options string that must consist of characters `slnfa`; unlike Lua that returns a table, the function returns all requested values one after another to improve performance.

New logo

Luau now has a shiny new logo!

!["New logo!"]({{ site.url }}{{ site.baseurl }}/assets/images/luau.png)

Coming soon...

- Generic variadics!
- Native Vector3 math with dramatic performance improvements!
- Better tools for memory analysis!
- Better treatment of cyclic requires during type checking!
- Better type refinements including nil-ability checks, `and`/`or` and `IsA`!

---

layout: single title: "Luau Recap: April 2021"

Luau is our new language that you can read more about at [https://roblox.github.io/luau (https://roblox.github.io/luau)](https://roblox.github.io/luau). Another busy month in Luau with many performance improvements.

[Cross-posted to the [Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-april-2021/)](https://devforum.roblox.com/t/luau-recap-april-2021/).]

Editor features

Luau implementation now provides an internal API for type-aware autocomplete suggestions.

Roblox Studio will be the first user of this API and we plan for a new beta feature to come soon in addition to existing Luau-powered beta features like Go To Declaration, Type Hovers and Script Function Filter (you should check those out!)

Performance improvements

Performance is a very important part of Luau implementation and we continue bringing in new performance optimizations:

- We've finished the work on internal `vector` value type that will be used by `Vector3` type in Roblox. Improvements of up to 10x can be seen for primitive operations and some of our heavy `Vector3` benchmarks have seen 2-3x improvement. You can read more about this feature [on Roblox Developer forums (https://devforum.roblox.com/t/native-luau-vector3-beta/)](https://devforum.roblox.com/t/native-luau-vector3-beta/)
- By optimizing the way string buffers are handled internally, we bring improvements to string operations including `string.lower`, `string.upper`, `string.reverse`, `string.rep`, `table.concat` and string concatenation operator `..`. Biggest improvements can be seen on large strings
- Improved performance of `table.insert` and `table.remove`. Operations in the middle of large arrays can be multiple times faster with this change
- Improved performance of internal table resize which brings additional 30% speedup for `table.insert`
- Improved performance of checks for missing table fields

Generic functions

We had to temporarily disable generic function definitions last month after finding critical issues in the implementation.

While they are still not available, we are making steady progress on fixing those issues and making additional typechecking improvements to bring them back in.

Debugger improvements

Debugging is now supported for parallel Luau Actors in Roblox Studio.

Read more about the feature on Roblox Developer forums (https://devforum.roblox.com/t/parallel-lua-beta/) and try it out yourself.

Behavior changes

Backwards compatibility is important for Luau, but sometimes a change is required to fix corner cases in the language / libraries or to improve performance. Even still, we try to keep impact of these changes to a minimum:

- __eq tag method will always get called for table comparisons even when a table is compared to itself

Coming soon...

- Better type refinements for statements under a condition using a new constraint resolver. Luau will now understand complex conditions combining `and`/`not` and type guards with more improvements to come

---

layout: single title: "Luau Recap: May 2021"

Luau is our new language that you can read more about at https://roblox.github.io/luau (https://roblox.github.io/luau). This month we have added a new small feature to the language and spent a lot of time improving our typechecker.

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-may-2021/).]

Named function type arguments

We've updated Luau syntax to support optional names of arguments inside function types. The syntax follows the same format as regular function argument declarations: `(a: number, b: string)`

Names can be provided in any place where function type is used, for example:

- in type aliases:

```
type MyCallbackType = (cost: number, name: string) -> string
```

- for variables:

```
local cb: (amount: number) -> number
local function foo(cb: (name: string) -> ())
```

Variadic arguments cannot have an extra name, they are already written as ...: number.

These names are used for documentation purposes and we also plan to display them in Roblox Studio auto-complete and type hovers. They do not affect how the typechecking of Luau scripts is performed.

Typechecking improvements

Speaking of typechecking, we've implemented many improvements this month:

- Typechecker will now visit bodies of all member functions, previously it didn't check methods if the self type was unknown
- Made improvements to cyclic module import detection and error reporting
- Fixed incorrect error on modification of table intersection type fields
- When using an 'or' between a nillable type and a value, the resulting type is now inferred to be non-nil
- We have improved error messages that suggest to use ':' for a method call
- Fixed order of types in type mismatch error that was sometimes reversed
- Fixed an issue with `table.insert` function signature

- Fixed a bug which caused spurious unknown global errors

We've also added new checks to our linter:

- A new check will report uses of deprecated Roblox APIs
- Linter will now suggest replacing globals with locals in more cases
- New warning is generated if array loop starts or ends on index '0', but the array is indexed from '1'
- FormatString lint will now check string patterns for `find`/`match` calls via `:` when object type is known to be a string

We also fixed one of the sources for "Free types leaked into this module's public interface" error message and we are working to fix the remaining ones.

As usual, typechecking improvements will not break execution of your games even if new errors get reported.

Editor features

We continue to improve our built-in support for auto-complete that will be used in future Roblox Studio updates and will make it easier to implement custom extensions for applications that support Language Server Protocol.

As part of this work we will improve the type information provided by Roblox APIs to match actual arguments and results.

Behavior changes

When a relational comparison fails at runtime, the error message now specifies the comparison direction (e.g. `attempt to compare nil <= number`)

Performance improvements

- Improved performance of table lookup with an index operator and a literal string: `t["name"]`
- Bytecode compilation is now ~5% faster which can improve server startup time for games with lots of scripts

layout: single title: "Luau Recap: June 2021"

Luau is our new language that you can read more about at https://roblox.github.io/luau (https://roblox.github.io/luau). Most of our team was busy working on improving Luau interaction with Roblox Studio for an upcoming feature this month, but we were able to add typechecking and performance improvements as well!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-june-2021/).]

Constraint Resolver

To improve type inference under conditional expressions and other dynamic type changes (like assignments) we have introduced a new constraint resolver framework into Luau type checker.

This framework allows us to handle more complex expressions that combine `and`/`not` operators and type guards.

Type guards support include expressions like:

- `if instance:IsA("ClassName") then`
- `if enum:IsA("EnumName") then`
- `if type(v) == "string" then`

This framework is extensible and we have plans for future improvements with `a == b`/`a ~= b` equality constraints and handling of table field assignments.

It is now also possible to get better type information inside `else` blocks of an `if` statement.

A few examples to see the constraint resolver in action:

```
function say_hello(name: string?)
    -- extra parentheses were enough to trip the old typechecker
    if (name) then
        print("Hello " .. name .. "!")
    else
        print("Hello mysterious stranger!")
    end
end
```

```
function say_hello(name: string?, surname: string?)
    -- but now we handle that and more complex expressions as well
    if not (name and surname) then
        print("Hello mysterious stranger!")
    else
        print("Hello " .. name .. " " .. surname .. "!")
    end
end
```

Please note that constraints are currently placed only on local and global variables. One of our goals is to include support for table members in the future.

Typechecking improvements

We have improved the way we handled module `require` calls. Previously, we had a simple pattern match on the `local m = require(...)` statement, but now we have replaced it with a general handling of the function call in any context.

Handling of union types in equality operators was fixed to remove incorrect error reports.

A new `IsA` method was introduced to EnumItem to check the type of a Roblox Enum. This is intended to replace the `enumItem.EnumType == Enum.NormalId` pattern in the code for a construct that allows our constraint resolver to infer better types.

Additional fixes include:

- `table.pack` return type was fixed
- A limit was added for deeply nested code blocks to avoid a crash
- We have improved the type names that are presented in error messages and Roblox Studio
- Error recovery was added to field access of a `table?` type. While you add a check for `nil`, typechecking can continue with better type information in other expressions.
- We handled a few internal compiler errors and rare crashes

Editor features

If you have Luau-Powered Type Hover beta feature enabled in Roblox Studio, you will see more function argument names inside function type hovers.

Behavior changes

We no longer allow referencing a function by name inside argument list of that function:

```
local function f(a: number, b: typeof(f)) -- 'f' is no longer visible here
```

Performance improvements

As always, we look for ways to improve performance of your scripts:

- We have fixed memory use of Roblox Actor scripts in Parallel Luau beta feature
- Performance of table clone through `table.move` has been greatly improved
- Table length lookup has been optimized, which also brings improvement to table element insertion speed
- Built-in Vector3 type support that we mentioned in April (https://devforum.roblox.com/t/native-luau-vector3-beta/) is now enabled for everyone

---

layout: single title: "Luau Recap: July 2021"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org). Our team was still busy working on upcoming Studio Beta feature for script editor, but we did fit in multiple typechecking improvements.

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-july-2021/).]

Typechecking improvements

A common complaint that we've received was a false-positive error when table with an optional or union element type is defined:

```
--!strict
type Foo = {x: number | string}
local foos: {Foo} = {
    {x = 1234567},
    {x = "hello"} -- Type 'string' could not be converted into 'number'
}
```

This case is now handled and skipping optional fields is allowed as well:

```
--!strict
type Foo = {
    a: number,
    b: number?
}
local foos: {Foo} = {
    { a = 1 },
    { a = 2, b = 3 } -- now ok
}
```

Current fix only handles table element type in assignments, but we plan to extend that to function call arguments and individual table fields.

Like we've mentioned last time, we will continue working on our new type constraint resolver and this month it learned to handle more complex expressions (including type guards) inside `assert` conditions:

```
--!strict
local part = script.Parent:WaitForChild("Part")
assert(part:IsA("BasePart"))
local basepart: BasePart = part -- no longer an error
```

And speaking of assertions, we applied a minor fix so that the type of the `assert` function correctly defines a second optional `string?` parameter.

We have also fixed the type of `string.gmatch` function reported by one of the community members. We know about issues in a few additional library functions and we'll work to fix them as well.

Hopefully, you didn't see 'free type leak' errors that underline your whole script, but some of you did and reported them to us. We read those reports and two additional cases have been fixed this month. We now track only a single one that should be fixed next month.

Another false positive error that was fixed involves tables with __call metatable function. We no longer report a type error when this method is invoked and we'll also make sure that given arguments match the function definition:

```
--!strict
local t = { x = 2 }

local x = setmetatable(t, {
    __call = function(self, a: number)
        return a * self.x
    end
})
local a = x(2) -- no longer an error
```

Please note that while call operator on a table is now handled, function types in Luau are distinct from table types and you'll still get an error if you try to assign this table to a variable of a function type.

Linter improvements

A new 'TableOperations' lint check was added that will detect common correctness or performance issues with `table.insert` and `table.remove`:

```
-- table.insert will insert the value before the last element, which is likely a bug; consider removing the second argument or wrap
table.insert(t, #t, 42)

-- table.insert will append the value to the table; consider removing the second argument for efficiency
table.insert(t, #t + 1, 42)

-- table.insert uses index 0 but arrays are 1-based; did you mean 1 instead?
table.insert(t, 0, 42)

-- table.remove uses index 0 but arrays are 1-based; did you mean 1 instead?
table.remove(t, 0)

-- table.remove will remove the value before the last element, which is likely a bug; consider removing the second argument or wrap
table.remove(t, #t - 1)

-- table.insert may change behavior if the call returns more than one result; consider adding parentheses around second argument
table.insert(t, string.find("hello", "h"))
```

Another new check is 'DuplicateConditions'. The name speaks for itself, `if` statement chains with duplicate conditions and expressions containing `and`/`or` operations with redundant parts will now be detected:

```
if x then
    -- ...
elseif not x then
    -- ...
elseif x then -- Condition has already been checked on line 1
    -- ...
end

local success = a and a -- Condition has already been checked on column 17

local good = (a or b) or a -- Condition has already been checked on column 15
```

We've also fixed an incorrect lint warning when `typeof` is used to check for `EnumItem`.

Editor features

An issue was fixed that prevented the debugger from displaying values inside Roblox callback functions when an error was reported inside of it.

Behavior changes

`table.insert` will no longer move elements forward 1 spot when index is negative or 0.

This change also fixed a performance issue when `table.insert` was called with a large negative index.

The 'TableOperations' lint mentioned earlier will flag cases where insertion at index 0 is performed.

---

layout: single title: "Luau Recap: August 2021"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-august-2021/).]

Editor features

The Roblox Studio Luau-Powered Autocomplete & Language Features Beta (https://devforum.roblox.com/t/script-editor-luau-powered-autocomplete-language-features-beta) that our team has been working on has finally been released! Be sure to check that out and leave your feedback for things we can improve.

To support that feature, a lot of work went into:

- Improving fault-tolerant parser recovery scenarios
- Storing additional information in the AST, including comments, better location information and partial syntax data
- Tracking additional information about types and their fields, including tracking definition locations, function argument names, deprecation state and custom Roblox-specific tags
- Updating reflection information to provide more specific `Instance` types and correct previously missing or wrong type annotations
- Hybrid typechecking mode which tries to infer types even in scripts with no typechecking enabled
- Support for types that are attached to the `DataModel` tree elements to provide instance member information
- Placing limits to finish typechecking in a finite space/time
- Adding Autocomplete API for the Roblox Studio to get location-based entity information and appropriate suggestions
- Additional type inference engine improvements and fixes

While our work continues to respond to the feedback we receive, our team members are shifting focus to add generic functions, improve type refinements in conditionals, extend Parallel Luau, improve Lua VM performance and provide documentation.

Typechecking improvements

Type constraint resolver now remembers constraints placed on individual table fields.

This should fix false-positive errors reported after making sure the optional table field is present:

```
--!strict
local t: {value: number?} = {value = 2}

if t.value then
    local v: number = t.value -- ok
end
```

And it can also refine field type to a more specific one:

```
--!strict
local t: {value: string|number} = {value = 2}

if type(t.value) == "number" then
    return t.value * 2 -- ok
end
```

Like before, combining multiple conditions using 'and' and 'not' is also supported.

Constructing arrays with different values for optional/union types are now also supported for individual table fields and in functions call arguments:

```
--!strict
type Foo = {x: number | string, b: number?}

local function foo(l: {Foo}) end

foo({
        {x = 1234567},
        {x = "hello"}, -- now ok
})

type Bar = {a: {Foo}}

local foos: Bar = {a = {
        {x = 1234567},
        {x = "hello", b = 2}, -- now ok
}}
```

Finally, we have fixed an issue with Roblox class field access using indexing like `part["Anchored"] = true`.

Linter improvements

We have added a new linter check for duplicate local variable definitions.

It is created to find duplicate names in cases like these:

```
local function foo(a1, a2, a2) -- Function argument 'a2' already defined on column 24
local a1, a2, a2 = f() -- Variable 'a2' already defined on column 11

local bar = {}
function bar:test(self) -- Function argument 'self' already defined implicitly
```

Our UnknownType linter warning was extended to check for correct class names passed into `FindFirstChildOfClass`, `FindFirstChildWhichIsA`, `FindFirstAncestorOfClass` and `FindFirstAncestorWhichIsA` functions.

Performance improvements

We have added an optimization to 'table.unpack' for 2x performance improvement.

We've also implemented an extra optimization for tables to predict required table capacity based on fields that are assigned to it in the code after construction. This can reduce the need to reallocate tables.

Variadic call performance was fine-tuned and is now ~10% faster.

Construction of array literals was optimized for a ~7% improvement.

Another optimization this month changes the location and rate of garbage collection invocations. We now try to avoid calling GC during the script execution and perform all the work in the GcJob part of the frame (it could be seen in the performance profiler). When possible, we can now skip that job in the frame completely, if we have some memory budget available.

Other improvements

For general stability improvements we fixed a crash when strange types like '`nil??`' are used and when users have their own global functions named '`require`'.

Indexing a table with an incompatible type will now show squiggly error lines under the index instead of the whole expression, which was a bit misleading.

An issue with debug information that caused `repeat ... until` condition line to be skipped when stepping was fixed.

Type output was improved to replace display of types like '`{<g405>(g405) -> g405}`' with '`{<a>(a) -> a}`'.

---

layout: single title: "Luau Recap: September 2021"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-september-2021/).]

Generic functions

The big news this month is that generic functions are back!

Luau has always supported type inference for generic functions, for example:

```
type Point<X,Y> = { x: X, y: Y }
function swap(p)
  return { x = p.y, y = p.x }
end
local p : Point<number, string> = swap({ x = "hi", y = 37 })
local q : Point<boolean, string> = swap({ x = "hi", y = true })
```

but up until now, there's been no way to write the type of `swap`, since Luau didn't have type parameters to functions (just regular old data parameters). Well, now you can:

```
function swap<X, Y>(p : Point<X, Y>): Point<Y, X>
  return { x = p.y, y = p.x }
end
```

Generic functions can be used in function declarations, and function types too, for example

```
type Swapper = { swap : <X, Y>(Point<X, Y>) -> Point<Y, X> }
```

People may remember that back in April (https://devforum.roblox.com/t/luau-recap-april-2021/) we announced generic functions, but then had to disable them. That was because DataBrain (https://devforum.roblox.com/u/databrain) discovered a nasty interaction (https://devforum.roblox.com/t/recent-type-system-regressions-for-generic-parametered-functions/) between `typeof` and generics, which meant that it was possible to write code that needed nested generic functions, which weren't supported back then.

Well, now we do support nested generic functions, so you can write code like

```
function mkPoint(x)
  return function(y)
    return { x = x, y = y }
  end
end
```

and have Luau infer a type where a generic function returns a generic function

```
function mkPoint<X>(x : X) : <Y>(Y) -> Point<X,Y>
  return function<Y>(y : Y) : Point<X,Y>
    return { x = x, y = y }
  end
end
```

For people who like jargon, Luau now supports *Rank N Types*, where previously it only supported Rank 1 Types.

Bidirectional typechecking

Up until now, Luau has used *bottom-up* typechecking. For example, for a function call `f(x)` we first find the type of `f` (say it's `(T)->U`) and the type for `x` (say it's `V`), make sure that `V` is a subtype of `T`, so the type of `f(x)` is `U`.

This works in many cases, but has problems with examples like registering callback event handlers. In code like

```
part.Touched:Connect(function (other) ... end)
```

if we try to typecheck this bottom-up, we have a problem because we don't know the type of `other` when we typecheck the body of the function.

What we want in this case is a mix of bottom-up and *top-down* typechecking. In this case, from the type of `part.Touched:Connect` we know that `other` must have type `BasePart`.

This mix of top-down and bottom-up typechecking is called *bidirectional typechecking*, and means that tools like type-directed autocomplete can provide better suggestions.

Editor features

We have made some improvements to the Luau-powered autocomplete beta feature in Roblox Studio:

- We no longer give autocomplete suggestions for client-only APIs in server-side scripts, or vice versa.
- For table literals with known shape, we provide autocomplete suggestions for properties.
- We provide autocomplete suggestions for `Player.PlayerGui`.
- Keywords such as `then` and `else` are autocompleted better.

- Autocompletion is disabled inside a comment span (a comment starting `--[[`).

Typechecking improvements

In other typechecking news:

- The Luau constraint resolver can now refine the operands of equality expressions.
- Luau type guard refinements now support more arbitrary cases, for instance `typeof(foo) ~= "Instance"` eliminates anything not a subclass of `Instance`.
- We fixed some crashes caused by use-after-free during type inference.
- We do a better job of tracking updates when script is moved inside the data model.
- We fixed one of the ways that recursive types could cause free types to leak (https://devforum.roblox.com/t/free-types-leaked-into-this-modules-public-interface/1459070).
- We improved the way that `return` statements interact with mutually recursive function declarations.
- We improved parser recovery from code which looks like a function call (but isn't) such as

```
local x = y
(expr)[smth] = z
```

- We consistently report parse errors before type errors.
- We display more types as `*unknown*` rather than as an internal type name like `error####`.
- Luau now infers the result of `Instance:Clone()` much more accurately.

Performance improvements

- `Vector3.new` constructor has been optimized and is now ~2x faster
- A previously implemented optimization for table size prediction has been enhanced to predict final table size when `setmetatable` is used, such as `local self = setmetatable({}, Klass)`
- Method calls for user-specified objects have been optimized and are now 2-4% faster
- `debug.traceback` is now 1.5x faster, although `debug.info` is likely still superior for performance-conscious code
- Creating table literals with explicit numeric indices, such as `{ [1] = 42 }`, is now noticeably faster, although list-style construction is still recommended.

Other improvements

- The existing 'TableLiteral' lint now flags cases when table literals have duplicate numeric indices, such as `{ [1] = 1, [1] = 2 }`

---

layout: single title: "Luau Recap: October 2021"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-october-2021/).]

if-then-else expression

In addition to supporting standard if *statements*, Luau adds support for if *expressions*. Syntactically, `if-then-else` expressions look very similar to if statements. However instead of conditionally executing blocks of code, if expressions conditionally evaluate expressions and return the value produced as a result. Also, unlike if statements, if expressions do not terminate with the `end` keyword.

Here is a simple example of an `if-then-else` expression:

```
local maxValue = if a > b then a else b
```

`if-then-else` expressions may occur in any place a regular expression is used. The `if-then-else` expression must match `if <expr> then <expr> else <expr>`; it can also contain an arbitrary number of `elseif` clauses, like `if <expr> then <expr> elseif <expr> then <expr> else <expr>`. Note that in either case, `else` is mandatory.

Here's is an example demonstrating `elseif`:

```
local sign = if x < 0 then -1 elseif x > 0 then 1 else 0
```

Note: In Luau, the `if-then-else` expression is preferred vs the standard Lua idiom of writing `a and b or c` (which roughly simulates a ternary operator). However, the Lua idiom may return an unexpected result if `b` evaluates to false. The `if-then-else` expression will behave as expected in all situations.

Library improvements

New additions to the `table` library have arrived:

```
function table.freeze(t)
```

Given a non-frozen table, freezes it such that all subsequent attempts to modify the table or assign its metatable raise an error. If the input table is already frozen or has a protected metatable, the function raises an error; otherwise it returns the input table. Note that the table is frozen in-place and is not being copied. Additionally, only `t` is frozen, and keys/values/metatable of `t` don't change their state and need to be frozen separately if desired.

```
function table.isfrozen(t): boolean
```

Returns `true` if and only if the input table is frozen.

Typechecking improvements

We continue work on our type constraint resolver and have multiple improvements this month.

We now resolve constraints that are created by `or` expressions. In the following example, by checking against multiple type alternatives, we learn that value is a union of those types:

```
--!strict
local function f(x: any)
    if type(x) == "number" or type(x) == "string" then
        local foo = x -- 'foo' type is known to be 'number | string' here
        -- ...
    end
end
```

Support for `or` constraints allowed us to handle additional scenarios with `and` and `not` expressions to reduce false positives after specific type guards.

And speaking of type guards, we now correctly handle sub-class relationships in those checks:

```
--!strict
local function f(x: Part | Folder | string)
    if typeof(x) == "Instance" then
        local foo = x -- 'foo' type is known to be 'Part | Folder' here
    else
        local bar = x -- 'bar' type is known to be 'string' here
    end
end
```

One more fix handles the `a and b or c` expression when 'b' depends on 'a':

```
--!strict
function f(t: {x: number}?)
    local a = t and t.x or 5 -- 'a' is a 'number', no false positive errors here
end
```

Of course, our new if-then-else expressions handle this case as well.

```
--!strict
function f(t: {x: number}?)
    local a = if t then t.x else 5 -- 'a' is a 'number', no false positive errors here
end
```

We have extended bidirectional typechecking that was announced last month to propagate types in additional statements and expressions.

```
--!strict
function getSortFunction(): (number, number) -> boolean
    return function(a, b) return a > b end -- a and b are now known to be 'number' here
end

local comp = getSortFunction()

comp = function(a, b) return a < b end -- a and b are now known to be 'number' here as well
```

We've also improved some of our messages with union types and optional types (unions types with `nil`).

When optional types are used incorrectly, you get better messages. For example:

```
--!strict
function f(a: {number}?)
    return a[1] -- "Value of type '{number}?' could be nil" instead of "'{number}?' is not a table'
end
```

When a property of a union type is accessed, but is missing from some of the options, we will report which options are not valid:

```
--!strict
type A = { x: number, y: number }
type B = { x: number }
local a: A | B
local b = a.y -- Key 'y' is missing from 'B' in the type 'A | B'
```

When we enabled generic functions last month, some users might have seen a strange error about generic functions not being compatible with regular ones.

This was caused by undefined behaviour of recursive types. We have now added a restriction on how generic type parameters can be used in recursive types: RFC: Recursive type restriction (https://github.com/Roblox/luau/blob/master/rfcs/recursive-type-restriction.md)

Performance improvements

An improvement to the Stop-The-World (atomic in Lua terms) stage of the garbage collector was made to reduce time taken by that step by 4x factor. While this step only happens once during a GC cycle, it cannot be split into small parts and long times were visible as frame time spikes.

Table construction and resize was optimized further; as a result, many instances of table construction see 10-20% improvements for smaller tables on all platforms and 20%+ improvements on Windows.

Bytecode compiler has been optimized for giant table literals, resulting in 3x higher compilation throughput for certain files on AMD Zen architecture.

Coroutine resumption has been optimized and is now ~10% faster for coroutine-heavy code.

Array reads and writes are also now a bit faster resulting in 1-3% lift in array-heavy benchmarks.

layout: single title: "Luau Goes Open-Source"

When Roblox was created 15 years ago, we chose Lua as the scripting language. Lua was small, fast, easy to embed and learn and opened up enormous possibilities for our developers.

A lot in Roblox was built on Lua including hundreds of thousands of lines of internally-developed code that powers Roblox App and Roblox Studio to this day, and the millions of experiences that developers have created. For many of them, it was the first programming language they've learned.

A few years ago, we started looking into how we can evolve Lua to be even faster, have better ergonomics, make it easier to write robust code and to unlock an ecosystem of rich tooling—from better static analysis to IDE integrations.

This is how Luau was born.

Luau is a new language that started from Lua 5.1 and kept evolving while keeping backwards compatibility and preserving the original design goals: simplicity, performance, embeddability.

We're incredibly grateful for the foundation that Lua has been—it's been a joy to build on top of! So now we want to give back to the community at large.

Starting today, Luau (https://luau-lang.org) is no longer an inseparable part of Roblox platform; it's a separate, open-source language.

Luau is available at https://github.com/Roblox/luau (https://github.com/Roblox/luau) and comes with the source code for the language runtime and all associated tooling: compiler, type checker, linter. The code is available to anyone, free of charge, under the terms of MIT License. We're happy to accept contributions to the language, whether that's documentation or source code.

The language evolution is driven by an RFC process that is also open to the public.

We are committed to improving Luau going forward—it remains a central piece of technology at Roblox. The team that works on the language keeps growing, and we have lots of ideas! The language will become even faster, even nicer to work with, even more powerful.

We can't wait to see what we can build, together.

layout: single title: "Luau Recap: November 2021"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-november-2021/).]

Type packs in type aliases

Type packs are the construct Luau uses to represent a sequence of types. We've had syntax for generic type packs for a while now, and it sees use in generic functions, but it hasn't been available in type aliases. That has changed, and it is now syntactically legal to write the following type alias:

```
type X<A...> = () -> A...
type Y = X<number, string>
```

We've also added support for explicit type packs. Previously, it was impossible to instantiate a generic with two or more type pack parameters, because it wasn't clear where the first pack ended and the second one began. We have introduced a new syntax for this use case:

```
type Fn<P..., R...> = (P...) -> R...
type X = Fn<(number, string), (string, number)>
```

For more information, check out the documentation (https://luau-lang.org/typecheck#type-packs) or the RFC (https://github.com/Roblox/luau/blob/f86d4c6995418e489a55be0100159009492778ff/rfcs/syntax-type-alias-type-packs.md) for this feature.

Luau is open-source!

We announced this in early November but it deserves repeating: Luau is now an open-source project! You can use Luau outside of Roblox, subject to MIT License, and - importantly - we accept contributions.

Many changes contributed by community, both Roblox and external, have been merged since we've made Luau open source. Of note are two visible changes that shipped on Roblox platform:

- The type error "Expected to return X values, but Y values are returned here" actually had X and Y swapped! This is now fixed.
- Luau compiler dutifully computed the length of the string when using `#` operator on a string literal; this is now fixed and `#"foo"` compiles to 3.

You might think that C++ is a scary language and you can't contribute to Luau. If so, you'd be happy to know that the contents of https://luau-lang.org (https://luau-lang.org), where we host our documentation, is also hosted on GitHub in the same repository (https://github.com/Roblox/luau/tree/master/docs (https://github.com/Roblox/luau/tree/master/docs)) and that we'd love the community to contribute improvements to documentation among other changes! For example see issues in this list (https://github.com/Roblox/luau/issues?q=is%3Aissue+is%3Aopen+label%3A%22pr+welcome%22) that start with "Documentation", but all other changes and additions to documentation are also welcome.

Library improvements

```
function bit32.countlz(n: number): number
function bit32.countrz(n: number): number
```

Given a number, returns the number of preceding left or trailing right-hand bits that are `0`.

See the RFC for these functions (https://github.com/Roblox/luau/blob/f86d4c6995418e489a55be0100159009492778ff/rfcs/function-bit32-countlz-countrz.md) for more information.

Type checking improvements

We have enabled a rewrite of how Luau handles `require` tracing. This has two main effects: firstly, in strict mode, `require` statements that Luau can't resolve will trigger type errors; secondly, Luau now understands the `FindFirstAncestor` method in `require` expressions.

Luau now warns when the index to `table.move` is 0, as this is non-idiomatic and performs poorly. If this behavior is intentional, wrap the index in parentheses to suppress the warning.

Luau now provides additional context in table and class type mismatch errors.

Performance improvements

We have enabled several changes that aim to avoid allocating a new closure object in cases where it's not necessary to. This is helpful in cases where many closures are being allocated; in our benchmark suite, the two benchmarks that allocate a large number of closures improved by 15% and 5%, respectively.

When checking union types, we now try possibilities whose synthetic names match. This will speed up type checking unions in cases where synthetic names are populated.

We have also enabled an optimization that shares state in a hot path on the type checker. This will improve type checking performance.

The Luau VM now attempts to cache the length of tables' array portion. This change showed a small performance improvement in benchmarks, and should speed up `#` expressions.

The Luau type checker now caches a specific category of table unification results. This can improve type checking performance significantly when the same set of types is used frequently.

When Luau is not retaining type graphs, the type checker now discards more of a module's type surface after type checking it. This improves memory usage significantly.

Bug fixes

We've fixed a bug where on ARM systems (mobile), packing negative numbers using unsigned formats in `string.pack` would produce the wrong result.

We've fixed an issue with type aliases that reuse generic type names that caused them to be instantiated incorrectly.

We've corrected a subtle bug that could cause free types to leak into a table type when a free table is bound to that table.

We've fixed an issue that could cause Luau to report an infinitely recursive type error when the type was not infinitely recursive.

---

layout: single title: "Luau Recap: January 2022"

Luau is our programming language that you can read more about at https://luau-lang.org (https://luau-lang.org).

Find us on GitHub (https://github.com/Roblox/luau)!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-january-2022/).]

Performance improvements

The implementation of `tostring` has been rewritten. This change replaces the default number->string conversion with a new algorithm called Schubfach, which allows us to produce the shortest precise round-trippable representation of any input number very quickly.

While performance is not the main driving factor, this also happens to be significantly faster than our old implementation (up to 10x depending on the number and the platform).

Make `tonumber(x)` ~2x faster by avoiding reparsing string arguments.

The Luau compiler now optimizes table literals where keys are constant variables the same way as if they were constants, eg

```
local r, g, b = 1, 2, 3
local col = { [r] = 255, [g] = 0, [b] = 255 }
```

Improvements to type assertions

The `::` type assertion operator can now be used to coerce a value between any two related types. Previously, it could only be used for downcasts or casts to `any`. The following used to be invalid, but is now valid:

```
local t = {x=0, y=0}
local a = t :: {x: number}
```

Typechecking improvements

An issue surrounding table literals and indexers has been fixed:

```
type RecolorMap = {[string]: RecolorMap | Color3}

local hatRecolorMap: RecolorMap = {
    Brim = Color3.fromRGB(255, 0, 0), -- We used to report an error here
    Top = Color3.fromRGB(255, 0, 0)
}
```

Accessing a property whose base expression was previously refined will now return the correct result.

Linter improvements

`table.create(N, {})` will now produce a static analysis warning since the element is going to be shared for all table entries.

Error reporting improvements

When a type error involves a union (or an option), we now provide more context in the error message.

For instance, given the following code:

```
--!strict

type T = {x: number}

local x: T? = {w=4}
```

We now report the following:

```
Type 'x' could not be converted into 'T?'
caused by:
  None of the union options are compatible. For example: Table type 'x' not compatible with type 'T' because the former is missing f
```

Luau now gives up and reports an `*unknown*` type in far fewer cases when typechecking programs that have type errors.

New APIs

We have brought in the `coroutine.close` (https://luau-lang.org/library#coroutine-library) function from Lua 5.4. It accepts a suspended coroutine and marks it as non-runnable. In Roblox, this can be useful in combination with `task.defer` to implement cancellation.

REPL improvements

The `luau` REPL application can be compiled from source or downloaded from releases page (https://github.com/Roblox/luau/releases). It has grown some new features:

- Added `--interactive` option to run the REPL after running the last script file.
- Allowed the compiler optimization level to be specified.
- Allowed methods to be tab completed
- Allowed methods on string instances to be completed
- Improved Luau REPL argument parsing and error reporting
- Input history is now saved/loaded

Thanks

A special thanks to all the fine folks who contributed PRs over the last few months!

- Halalaluyafail3 (https://github.com/Halalaluyafail3)
- JohnnyMorganz (https://github.com/JohnnyMorganz)
- Kampfkarren (https://github.com/Kampfkarren)
- kunitoki (https://github.com/kunitoki)
- MathematicalDessert (https://github.com/MathematicalDessert)
- metatablecat (https://github.com/metatablecat)
- petrihakkinen (https://github.com/petrihakkinen)
- rafa_br34 (https://github.com/rafa_br34)
- Rerumu (https://github.com/Rerumu)
- Slappy826 (https://github.com/Slappy826)
- SnowyShiro (https://github.com/SnowyShiro)
- vladmarica (https://github.com/vladmarica)
- xgladius (https://github.com/xgladius)

Contribution guide (https://github.com/Roblox/luau/blob/2f989fc049772f36de1a4281834c375858507bda/CONTRIBUTING.md)

layout: single title: "Luau Recap: February 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-february-2022/).]

Default type alias type parameters

We have introduced a syntax to provide default type arguments inside the type alias type parameter list.

It is now possible to have type functions where the instantiation can omit some type arguments.

You can provide concrete types:

```
--!strict
type FieldResolver<T, Data = {[string]: any}> = (T, Data) -> number

local a: FieldResolver<number> = ...
local b: FieldResolver<number, {name: string}> = ...
```

Or reference parameters defined earlier in the list:

```
--!strict
type EqComp<T, U = T> = (l: T, r: U) -> boolean

local a: EqComp<number> = ... -- (l: number, r: number) -> boolean
local b: EqComp<number, string> = ... -- (l: number, r: string) -> boolean
```

Type pack parameters can also have a default type pack:

```
--!strict
type Process<T, U... = ...string> = (T) -> U...

local a: Process<number> = ... -- (number) -> ...string
local b: Process<number, (boolean, string)> = ... -- (number) -> (boolean, string)
```

If all type parameters have a default type, it is now possible to reference that without providing any type arguments:

```
--!strict
type All<T = string, U = number> = (T) -> U

local a: All -- ok
local b: All<> -- ok as well
```

For more details, you can read the original RFC proposal (https://github.com/Roblox/luau/blob/master/rfcs/syntax-default-type-alias-type-parameters.md).

Typechecking improvements

This month we had many fixes to improve our type inference and reduce false positive errors.

if-then-else expression can now have different types in each branch:

```
--!strict
local a = if x then 5 else nil -- 'a' will have type 'number?'
local b = if x then 1 else '2' -- 'b' will have type 'number | string'
```

And if the expected result type is known, you will not get an error in cases like these:

```
--!strict
type T = {number | string}
-- different array element types don't give an error if that is expected
local c: T = if x then {1, "x", 2, "y"} else {0}
```

`assert` result is now known to not be 'falsy' (`false` or `nil`):

```
--!strict
local function f(x: number?): number
    return assert(x) -- no longer an error
end
```

We fixed cases where length operator `#` reported an error when used on a compatible type:

```
--!strict
local union: {number} | {string}
local a = #union -- no longer an error
```

Functions with different variadic argument/return types are no longer compatible:

```
--!strict
local function f(): (number, ...string)
    return 2, "a", "b"
end

local g: () -> (number, ...boolean) = f -- error
```

We have also fixed:

- false positive errors caused by incorrect reuse of generic types across different function declarations
- issues with forward-declared intersection types
- wrong return type annotation for table.move
- various crashes reported by developers

Linter improvements

A new static analysis warning was introduced to mark incorrect use of a '`a and b or c`' pattern. When 'b' is 'falsy' (`false` or `nil`), result will always be 'c', even if the expression 'a' was true:

```
local function f(x: number)
    -- The and-or expression always evaluates to the second alternative because the first alternative is false; consider using if-the
    return x < 0.5 and false or 42
end
```

Like we say in the warning, new if-then-else expression doesn't have this pitfall:

```
local function g(x: number)
    return if x < 0.5 then false else 42
end
```

We have also introduced a check for misspelled comment directives:

```
--!non-strict
-- ^ Unknown comment directive 'non-strict'; did you mean 'nonstrict'?
```

Performance improvements

For performance, we have changed how our Garbage Collector collects unreachable memory.
This rework makes it possible to free memory 2.5x faster and also comes with a small change to how we store Luau objects in memory. For example, each table now uses 16 fewer bytes on 64-bit platforms.

Another optimization was made for `select(_, ...)` call.
It is now using a special fast path that has constant-time complexity in number of arguments (~3x faster with 10 arguments).

Thanks

A special thanks to all the fine folks who contributed PRs this month!

- mikejsavage (https://github.com/mikejsavage)
- TheGreatSageEqualToHeaven (https://github.com/TheGreatSageEqualToHeaven)
- petrihakkinen (https://github.com/petrihakkinen)

layout: single title: "Luau Recap: March 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-march-2022/).]

Singleton types

We added support for singleton types! These allow you to use string or boolean literals in types. These types are only inhabited by the literal, for example if a variable `x` has type `"foo"`, then `x == "foo"` is guaranteed to be true.

Singleton types are particularly useful when combined with union types, for example:

```
type Animals = "Dog" | "Cat" | "Bird"
```

or:

```
type Falsey = false | nil
```

In particular, singleton types play well with unions of tables, allowing tagged unions (also known as discriminated unions):

```
type Ok<T> = { type: "ok", value: T }
type Err<E> = { type: "error", error: E }
type Result<T, E> = Ok<T> | Err<E>

local result: Result<number, string> = ...
if result.type == "ok" then
    -- result :: Ok<number>
    print(result.value)
elseif result.type == "error" then
    -- result :: Err<string>
    error(result.error)
end
```

The RFC for singleton types is https://github.com/Roblox/luau/blob/master/rfcs/syntax-singleton-types.md (https://github.com/Roblox/luau/blob/master/rfcs/syntax-singleton-types.md)

Width subtyping

A common idiom for programming with tables is to provide a public interface type, but to keep some of the concrete implementation private, for example:

```
type Interface = {
    name: string,
}

type Concrete = {
    name: string,
    id: number,
}
```

Within a module, a developer might use the concrete type, but export functions using the interface type:

```
local x: Concrete = {
    name = "foo",
    id = 123,
}

local function get(): Interface
    return x
end
```

Previously examples like this did not typecheck but now they do!

This language feature is called *width subtyping* (it allows tables to get *wider*, that is to have more properties).

The RFC for width subtyping is https://github.com/Roblox/luau/blob/master/rfcs/sealed-table-subtyping.md (https://github.com/Roblox/luau/blob/master/rfcs/sealed-table-subtyping.md)

Typechecking improvements

- Generic function type inference now works the same for generic types and generic type packs.
- We improved some error messages.
- There are now fewer crashes (hopefully none!) due to mutating types inside the Luau typechecker.
- We fixed a bug that could cause two incompatible copies of the same class to be created.
- Luau now copes better with cyclic metatable types (it gives a type error rather than hanging).
- Fixed a case where types are not properly bound to all of the subtype when the subtype is a union.
- We fixed a bug that confused union and intersection types of table properties.
- Functions declared as `function f(x : any)` can now be called as `f()` without a type error.

API improvements

- Implement `table.clone` which takes a table and returns a new table that has the same keys/values/metatable. The cloning is shallow - if some keys refer to tables that need to be cloned, that can be done manually by modifying the resulting table.

Debugger improvements

- Use the property name as the name of methods in the debugger.

Performance improvements

- Optimize table rehashing (~15% faster dictionary table resize on average)
- Improve performance of freeing tables (~5% lift on some GC benchmarks)
- Improve gathering performance metrics for GC.
- Reduce stack memory reallocation.

---

layout: single title: "Luau Recap: April 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-april-2022/).]

It's been a bit of a quiet month. We mostly have small optimizations and bugfixes for you.

It is now allowed to define functions on sealed tables that have string indexers. These functions will be typechecked against the indexer type. For example, the following is now valid:

```
local a : {[string]: () -> number} = {}

function a.y() return 4 end -- OK
```

Autocomplete will now provide string literal suggestions for singleton types. eg

```
local function f(x: "a" | "b") end
f("_") -- suggest "a" and "b"
```

Improve error recovery in the case where we encounter a type pack variable in a place where one is not allowed. eg `type Foo<A...> = { value: A... }`

When code does not pass enough arguments to a variadic function, the error feedback is now better.

For example, the following script now produces a much nicer error message:

```
type A = { [number]: number }
type B = { [number]: string }

local a: A = { 1, 2, 3 }

-- ERROR: Type 'A' could not be converted into 'B'
--     caused by:
--       Property '[indexer value]' is not compatible. Type 'number' could not be converted into 'string'
local b: B = a
```

If the following code were to error because `Hello` was undefined, we would erroneously include the comment in the span of the error. This is now fixed.

```
type Foo = Hello -- some comment over here
```

Fix a crash that could occur when strict scripts have cyclic require() dependencies.

Add an option to autocomplete to cause it to abort processing after a certain amount of time has elapsed.

layout: single title: "Luau Recap: May 2022"

This month Luau team has worked to bring you a new language feature together with more typechecking improvements and bugfixes!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-may-2022/).]

Generalized iteration

We have extended the semantics of standard Lua syntax for iterating through containers, `for vars in values` with support for generalized iteration.
In Lua, to iterate over a table you need to use an iterator like `next` or a function that returns one like `pairs` or `ipairs`. In Luau, you can now simply iterate over a table:

```
for k, v in {1, 4, 9} do
    assert(k * k == v)
end
```

This works for tables but can also be customized for tables or userdata by implementing `__iter` metamethod. It is called before the iteration begins, and should return an iterator function like `next` (or a custom one):

```
local obj = { items = {1, 4, 9} }
setmetatable(obj, { __iter = function(o) return next, o.items end })

for k, v in obj do
    assert(k * k == v)
end
```

The default iteration order for tables is specified to be consecutive for elements `1..#t` and unordered after that, visiting every element.
Similar to iteration using `pairs`, modifying the table entries for keys other than the current one results in unspecified behavior.

Typechecking improvements

We have added a missing check to compare implicit table keys against the key type of the table indexer:

```
-- error is correctly reported, implicit keys (1,2,3) are not compatible with [string]
local t: { [string]: boolean } = { true, true, false }
```

Rules for `==` and `~=` have been relaxed for union types, if any of the union parts can be compared, operation succeeds:

```
--!strict
local function compare(v1: Vector3, v2: Vector3?)
    return v1 == v2 -- no longer an error
end
```

Table value type propagation now correctly works with `[any]` key type:

```
--!strict
type X = {[any]: string | boolean}
local x: X = { key = "str" } -- no longer gives an incorrect error
```

If a generic function doesn't provide type annotations for all arguments and the return value, additional generic type parameters might be added automatically:

```
-- previously it was foo<T>, now it's foo<T, b>, because second argument is also generic
function foo<T>(x: T, y) end
```

We have also fixed various issues that have caused crashes, with many of them coming from your bug reports.

Linter improvements

`GlobalUsedAsLocal` lint warning has been extended to notice when global variable writes always happen before their use in a local scope, suggesting that they can be replaced with a local variable:

```
function bar()
    foo = 6 -- Global 'foo' is never read before being written. Consider changing it to local
    return foo
end
function baz()
    foo = 10
    return foo
end
```

Performance improvements

Garbage collection CPU utilization has been tuned to further reduce frame time spikes of individual collection steps and to bring different GC stages to the same level of CPU utilization.

Returning a type-cast local (`return a :: type`) as well as returning multiple local variables (`return a, b, c`) is now a little bit more efficient.

Function inlining and loop unrolling

In the open-source release of Luau, when optimization level 2 is enabled, the compiler will now perform function inlining and loop unrolling.

Only loops with loop bounds known at compile time, such as `for i=1,4 do`, can be unrolled. The loop body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if unrolling should be performed.

Only local functions (defined either as `local function foo` or `local foo = function`) can be inlined. The function body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if each call to the function should be inlined instead. Additionally recursive invocations of a function can't be inlined at this time, and inlining is completely disabled for modules that use `getfenv`/`setfenv` functions.

---

layout: single title: "Luau Recap: June 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-june-2022/).]

Lower bounds calculation

A common problem that Luau has is that it primarily works by inspecting expressions in your program and narrowing the *upper bounds* of the values that can inhabit particular variables. In other words, each time we see a variable used, we eliminate possible sets of values from that variable's domain.

There are some important cases where this doesn't produce a helpful result. Take this function for instance:

```
function find_first_if(vec, f)
        for i, e in ipairs(vec) do
                if f(e) then
                        return i
                end
        end

        return nil
end
```

Luau scans the function from top to bottom and first sees the line `return i`. It draws from this the inference that `find_first_if` must return the type of `i`, namely `number`.

This is fine, but things go sour when we see the line `return nil`. Since we are always narrowing, we take from this line the judgement that the return type of the function is `nil`. Since we have already concluded that the function must return `number`, Luau reports an error.

What we actually want to do in this case is to take these `return` statements as inferences about the *lower* bound of the function's return type. Instead of saying "this function must return values of type `nil`," we should instead say "this function may *also* return values of type `nil`."

Lower bounds calculation does precisely this. Moving forward, Luau will instead infer the type `number?` for the above function.

This does have one unfortunate consequence: If a function has no return type annotation, we will no longer ever report a type error on a `return` statement. We think this is the right balance but we'll be keeping an eye on things just to be sure.

Lower-bounds calculation is larger and a little bit riskier than other things we've been working on so we've set up a beta feature in Roblox Studio to enable them. It is called "Experimental Luau language features."

Please try it out and let us know what you think!

Known bug

We have a known bug with certain kinds of cyclic types when lower-bounds calculation is enabled. The following, for instance, is known to be problematic.

```
type T = {T?}? -- spuriously reduces to {nil}?
```

We hope to have this fixed soon.

All table literals now result in unsealed tables

Previously, the only way to create a sealed table was by with a literal empty table. We have relaxed this somewhat: Any table created by a `{}` expression is considered to be unsealed within the scope where it was created:

```
local T = {}
T.x = 5 -- OK

local V = {x=5}
V.y = 2 -- previously disallowed.  Now OK.

function mkTable()
    return {x = 5}
end

local U = mkTable()
U.y = 2 -- Still disallowed: U is sealed
```

Other fixes

- Adjust indentation and whitespace when creating multiline string representations of types, resulting in types that are easier to read.
- Some small bugfixes to autocomplete
- Fix a case where accessing a nonexistent property of a table would not result in an error being reported.
- Improve parser recovery for the incorrect code `function foo() -> ReturnType` (the correct syntax is `function foo(): ReturnType`)
- Improve the parse error offered for code that improperly uses the `function` keyword to start a type eg `type T = function`
- Some small crash fixes and performance improvements

Thanks!

A very special thanks to all of our open source contributors:

- Allan N Jeremy (https://github.com/AllanJeremy)
- Daniel Nachun (https://github.com/danielnachun)
- JohnnyMorganz (https://github.com/JohnnyMorganz/)
- Petri Häkkinen (https://github.com/petrihakkinen)
- Qualadore (https://github.com/Qualadore)

layout: single title: "Luau Recap: July & August 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-july-august-2022/).]

Tables now support `__len` metamethod

See the RFC Support `__len` metamethod for tables and `rawlen` function (https://github.com/Roblox/luau/blob/master/rfcs/len-metamethod-rawlen.md) for more details.

With generalized iteration released in May, custom containers are easier than ever to use. The only thing missing was the fact that tables didn't respect `__len`.

Simply, tables now honor the `__len` metamethod, and `rawlen` is also added with similar semantics as `rawget` and `rawset`:

```
local my_cool_container = setmetatable({ items = { 1, 2 } }, {
    __len = function(self) return #self.items end
})

print(#my_cool_container) --> 2
print(rawlen(my_cool_container)) --> 0
```

`never` and `unknown` types

See the RFC `never` and `unknown` types (https://github.com/Roblox/luau/blob/master/rfcs/never-and-unknown-types.md) for more details.

We've added two new types, `never` and `unknown`. These two types are the opposites of each other by the fact that there's no value that inhabits the type `never`, and the dual of that is every value inhabits the type `unknown`.

Type inference may infer a variable to have the type `never` if and only if the set of possible types becomes empty, for example through type refinements.

```
function f(x: string | number)
    if typeof(x) == "string" and typeof(x) == "number" then
        -- x: never
    end
end
```

This is useful because we still needed to ascribe a type to `x` here, but the type we used previously had unsound semantics. For example, it was possible to be able to *expand* the domain of a variable once the user had proved it impossible. With `never`, narrowing a type from `never` yields `never`.

Conversely, `unknown` can be used to enforce a stronger contract than `any`. That is, `unknown` and `any` are similar in terms of allowing every type to inhabit them, and other than `unknown` or `any`, `any` allows itself to inhabit into a different type, whereas `unknown` does not.

```
function any(): any return 5 end
function unknown(): unknown return 5 end

-- no type error, but assigns a number to x which expects string
local x: string = any()

-- has type error, unknown cannot be converted into string
local y: string = unknown()
```

To be able to do this soundly, you must apply type refinements on a variable of type `unknown`.

```
local u = unknown()

if typeof(u) == "string" then
    local y: string = u -- no type error
end
```

A use case of `unknown` is to enforce type safety at implementation sites for data that do not originate in code, but from over the wire.

Argument names in type packs when instantiating a type

We had a bug in the parser which erroneously allowed argument names in type packs that didn't fold into a function type. That is, the below syntax did not generate a parse error when it should have.

```
Foo<(a: number, b: string)>
```

New IntegerParsing lint

See the announcement (https://devforum.roblox.com/t/improving-binary-and-hexadecimal-integer-literal-parsing-rules-in-luau/) for more details. We include this here for posterity.

We've introduced a new lint called IntegerParsing. Right now, it lints three classes of errors:

1. Truncation of binary literals that resolves to a value over 64 bits,
2. Truncation of hexadecimal literals that resolves to a value over 64 bits, and
3. Double hexadecimal prefix.

For 1.) and 2.), they are currently not planned to become a parse error, so action is not strictly required here.

For 3.), this will be a breaking change! See the rollout plan (https://devforum.roblox.com/t/improving-binary-and-hexadecimal-integer-literal-parsing-rules-in-luau/#rollout-5) for details.

New ComparisonPrecedence lint

We've also introduced a new lint called `ComparisonPrecedence`. It fires in two particular cases:

1. `not X op Y` where `op` is `==` or `~=`, or
2. `X op Y op Z` where `op` is any of the comparison or equality operators.

In languages that uses `!` to negate the boolean i.e. `!x == y` looks fine because `!x` *visually* binds more tightly than Lua's equivalent, `not x`. Unfortunately, the precedences here are identical, that is `!x == y` is `(!x) == y` in the same way that `not x == y` is `(not x) == y`. We also apply this on other operators e.g. `x <= y == y`.

```
-- not X == Y is equivalent to (not X) == Y; consider using X ~= Y, or wrap one of the expressions in parentheses to silence
if not x == y then end

-- not X ~= Y is equivalent to (not X) ~= Y; consider using X == Y, or wrap one of the expressions in parentheses to silence
if not x ~= y then end

-- not X <= Y is equivalent to (not X) <= Y; wrap one of the expressions in parentheses to silence
if not x <= y then end

-- X <= Y == Z is equivalent to (X <= Y) == Z; wrap one of the expressions in parentheses to silence
if x <= y == 0 then end
```

As a special exception, this lint pass will not warn for cases like `x == not y` or `not x == not y`, which both looks intentional as it is written and interpreted.

Function calls returning singleton types incorrectly widened

Fix a bug where widening was a little too happy to fire in the case of function calls returning singleton types or union thereof. This was an artifact of the logic that knows not to infer singleton types in cases that makes no sense to.

```
function f(): "abc" | "def"
    return if math.random() > 0.5 then "abc" else "def"
end

-- previously reported that 'string' could not be converted into '"abc" | "def"'
local x: "abc" | "def" = f()
```

`string` can be a subtype of a table with a shape similar to `string`

The function `my_cool_lower` is a function `<a...>(t: t1) -> a... where t1 = {+ lower: (t1) -> a... +}`.

```
function my_cool_lower(t)
    return t:lower()
end
```

Even though `t1` is a table type, we know `string` is a subtype of `t1` because `string` also has `lower` which is a subtype of `t1`'s `lower`, so this call site now type checks.

```
local s: string = my_cool_lower("HI")
```

Other analysis improvements

- `string.gmatch`/`string.match`/`string.find` may now return more precise type depending on the patterns used
- Fix a bug where type arena ownership invariant could be violated, causing stability issues
- Fix a bug where internal type error could be presented to the user
- Fix a false positive with optionals & nested tables
- Fix a false positive in non-strict mode when using generalized iteration
- Improve autocomplete behavior in certain cases for `:` calls
- Fix minor inconsistencies in synthesized names for types with metatables
- Fix autocomplete not suggesting globals defined after the cursor
- Fix DeprecatedGlobal warning text in cases when the global is deprecated without a suggested alternative
- Fix an off-by-one error in type error text for incorrect use of `string.format`

Other runtime improvements

- Comparisons with constants are now significantly faster when using clang as a compiler (10-50% gains on internal benchmarks)
- When calling non-existent methods on tables or strings, `foo:bar` now produces a more precise error message
- Improve performance for iteration of tables
- Fix a bug with negative zero in vector components when using vectors as table keys
- Compiler can now constant fold builtins under -O2, for example `string.byte("A")` is compiled to a constant
- Compiler can model the cost of builtins for the purpose of inlining/unrolling
- Local reassignment i.e. `local x = y :: T` is free iff neither `x` nor `y` is mutated/captured
- Improve `debug.traceback` performance by 1.15-1.75x depending on the platform
- Fix a corner case with table assignment semantics when key didn't exist in the table and `__newindex` was defined: we now use Lua 5.2 semantics and call `__newindex`, which results in less wasted space, support for NaN keys in `__newindex` path and correct support for frozen tables
- Reduce parser C stack consumption which fixes some stack overflow crashes on deeply nested sources
- Improve performance of `bit32.extract`/`replace` when width is implied (~3% faster chess)
- Improve performance of `bit32.extract` when field/width are constants (~10% faster base64)
- `string.format` now supports a new format specifier, `%*`, that accepts any value type and formats it using `tostring` rules

Thanks

Thanks for all the contributions!

- natteko (https://github.com/natteko)
- JohnnyMorganz (https://github.com/JohnnyMorganz)
- khvzak (https://github.com/khvzak)
- Anaminus (https://github.com/Anaminus)
- memery-rbx (https://github.com/memery-rbx)
- jaykru (https://github.com/jaykru)
- Kampfkarren (https://github.com/Kampfkarren)
- XmiliaH (https://github.com/XmiliaH)
- Mactavsin (https://github.com/Mactavsin)

---

layout: single title: "Semantic Subtyping in Luau" author: Alan Jeffrey

Luau is the first programming language to put the power of semantic subtyping in the hands of millions of creators.

Minimizing false positives

One of the issues with type error reporting in tools like the Script Analysis widget in Roblox Studio is *false positives*. These are warnings that are artifacts of the analysis, and don't correspond to errors which can occur at runtime. For example, the program

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

reports a type error which cannot happen at runtime, since `CFrame` supports multiplication by both `Vector3` and `CFrame`. (Its type is `((CFrame, CFrame) -> CFrame) & ((CFrame, Vector3) -> Vector3)`.)

False positives are especially poor for onboarding new users. If a type-curious creator switches on typechecking and is immediately faced with a wall of spurious red squiggles, there is a strong incentive to immediately switch it off again.

Inaccuracies in type errors are inevitable, since it is impossible to decide ahead of time whether a runtime error will be triggered. Type system designers have to choose whether to live with false positives or false negatives. In Luau this is determined by the mode: `strict` mode errs on the side of false positives, and `nonstrict` mode errs on the side of false negatives.

While inaccuracies are inevitable, we try to remove them whenever possible, since they result in spurious errors, and imprecision in type-driven tooling like autocomplete or API documentation.

Subtyping as a source of false positives

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()) 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 (`T & U`). For example, `number?` is implemented as a union type `(number | nil)`, inhabited by values that are either numbers or `nil`.

Unfortunately, the interaction of subtyping with intersection and union types can have odd results. A simple (but rather artificial) case in older Luau was:

```
local x : (number?) & (string?) = nil
local y : nil = nil
y = x -- Type '(number?) & (string?)' could not be converted into 'nil'
x = y
```

This error is caused by a failure of subtyping, the old subtyping algorithm reports that `(number?) & (string?)` is not a subtype of `nil`. This is a false positive, since `number & string` is uninhabited, so the only possible inhabitant of `(number?) & (string?)` is `nil`.

This is an artificial example, but there are real issues raised by creators caused by the problems, for example https://devforum.roblox.com/t/luau-recap-july-2021/1382101/5 (https://devforum.roblox.com/t/luau-recap-july-2021/1382101/5). Currently, these issues mostly affect creators making use of sophisticated type system features, but as we make type inference more accurate, union and intersection types will become more common, even in code with no type annotations.

This class of false positives no longer occurs in Luau, as we have moved from our old approach of *syntactic subtyping* to an alternative called *semantic subtyping*.

Syntactic subtyping

AKA “what we did before.”

Syntactic subtyping is a syntax-directed recursive algorithm. The interesting cases to deal with intersection and union types are:

- Reflexivity: `T` is a subtype of `T`
- Intersection L: `(`T_1 `& … &` T_j`)` is a subtype of `U` whenever some of the T_i are subtypes of `U`
- Union L: `(`T_1 `| … |` T_j`)` is a subtype of `U` whenever all of the T_i are subtypes of `U`
- Intersection R: `T` is a subtype of `(`U_1 `& … &` U_j`)` whenever `T` is a subtype of all of the U_i
- Union R: `T` is a subtype of `(`U_1 `| … |` U_j`)` whenever `T` is a subtype of some of the U_i.

For example:

- By Reflexivity: `nil` is a subtype of `nil`
- so by Union R: `nil` is a subtype of `number?`
- and: `nil` is a subtype of `string?`
- so by Intersection R: `nil` is a subtype of `(number?) & (string?).`

Yay! Unfortunately, using these rules:

- `number` isn't a subtype of `nil`
- so by Union L: `(number?)` isn't a subtype of `nil`
- and: `string` isn't a subtype of `nil`
- so by Union L: `(string?)` isn't a subtype of `nil`
- so by Intersection L: `(number?) & (string?)` isn't a subtype of `nil`.

This is typical of syntactic subtyping: when it returns a "yes" result, it is correct, but when it returns a "no" result, it might be wrong. The algorithm is a *conservative approximation*, and since a "no" result can lead to type errors, this is a source of false positives.

Semantic subtyping

AKA "what we do now."

Rather than thinking of subtyping as being syntax-directed, we first consider its semantics, and later return to how the semantics is implemented. For this, we adopt semantic subtyping:

- The semantics of a type is a set of values.
- Intersection types are thought of as intersections of sets.
- Union types are thought of as unions of sets.
- Subtyping is thought of as set inclusion.

For example:

Type	Semantics
`number`	{ 1, 2, 3, … }
`string`	{ "foo", "bar", … }
`nil`	
`number?`	{ nil, 1, 2, 3, … }
`string?`	{ nil, "foo", "bar", … }
`(number?) & (string?)`	{ nil, 1, 2, 3, … } ∩ { nil, "foo", "bar", … } =

and since subtypes are interpreted as set inclusions:

Subtype	Supertype	Because
`nil`	`number?`	⊆ { nil, 1, 2, 3, … }
`nil`	`string?`	⊆ { nil, "foo", "bar", … }
`nil`	`(number?) & (string?)`	⊆
`(number?) & (string?)`	`nil`	⊆

So according to semantic subtyping, `(number?) & (string?)` is equivalent to `nil`, but syntactic subtyping only supports one direction.

This is all fine and good, but if we want to use semantic subtyping in tools, we need an algorithm, and it turns out checking semantic subtyping is non-trivial.

Semantic subtyping is hard

NP-hard to be precise.

We can reduce graph coloring to semantic subtyping by coding up a graph as a Luau type such that checking subtyping on types has the same result as checking for the impossibility of coloring the graph

For example, coloring a three-node, two color graph can be done using types:

```
type Red = "red"
type Blue = "blue"
type Color = Red | Blue
type Coloring = (Color) -> (Color) -> (Color) -> boolean
type Uncolorable = (Color) -> (Color) -> (Color) -> false
```

Then a graph can be encoded as an overload function type with subtype `Uncolorable` and supertype `Coloring`, as an overloaded function which returns `false` when a constraint is violated. Each overload encodes one constraint. For example a line has constraints saying that adjacent nodes cannot have the same color:

```
type Line = Coloring
  & ((Red) -> (Red) -> (Color) -> false)
  & ((Blue) -> (Blue) -> (Color) -> false)
  & ((Color) -> (Red) -> (Red) -> false)
  & ((Color) -> (Blue) -> (Blue) -> false)
```

A triangle is similar, but the end points also cannot have the same color:

```
type Triangle = Line
  & ((Red) -> (Color) -> (Red) -> false)
  & ((Blue) -> (Color) -> (Blue) -> false)
```

Now, `Triangle` is a subtype of `Uncolorable`, but `Line` is not, since the line can be 2-colored. This can be generalized to any finite graph with any finite number of colors, and so subtype checking is NP-hard.

We deal with this in two ways:

- we cache types to reduce memory footprint, and
- give up with a “Code Too Complex” error if the cache of types gets too large.

Hopefully this doesn't come up in practice much. There is good evidence that issues like this don't arise in practice from experience with type systems like that of Standard ML, which is EXPTIME-complete (https://dl.acm.org/doi/abs/10.1145/96709.96748), but in practice you have to go out of your way to code up Turing Machine tapes as types.

Type normalization

The algorithm used to decide semantic subtyping is *type normalization*. Rather than being directed by syntax, we first rewrite types to be normalized, then check subtyping on normalized types.

A normalized type is a union of:

- a normalized nil type (either `never` or `nil`)
- a normalized number type (either `never` or `number`)
- a normalized boolean type (either `never` or `true` or `false` or `boolean`)
- a normalized function type (either `never` or an intersection of function types) etc

Once types are normalized, it is straightforward to check semantic subtyping.

Every type can be normalized (sigh, with some technical restrictions around generic type packs). The important steps are:

- removing intersections of mismatched primitives, e.g. `number & bool` is replaced by `never`, and
- removing unions of functions, e.g. `((number?) -> number) | ((string?) -> string)` is replaced by `(nil) -> (number | string)`.

For example, normalizing `(number?) & (string?)` removes `number & string`, so all that is left is `nil`.

Our first attempt at implementing type normalization applied it liberally, but this resulted in dreadful performance (complex code went from typechecking in less than a minute to running overnight). The reason for this is annoyingly simple: there is an optimization in Luau's subtyping algorithm to handle reflexivity (`T` is a subtype of `T`) that performs a cheap pointer equality check. Type normalization can convert pointer-identical types into semantically-equivalent (but not pointer-identical) types, which significantly degrades performance.

Because of these performance issues, we still use syntactic subtyping as our first check for subtyping, and only perform type normalization if the syntactic algorithm fails. This is sound, because syntactic subtyping is a conservative approximation to semantic subtyping.

Pragmatic semantic subtyping

Off-the-shelf semantic subtyping is slightly different from what is implemented in Luau, because it requires models to be *set-theoretic*, which requires that inhabitants of function types "act like functions." There are two reasons why we drop this requirement.

Firstly, we normalize function types to an intersection of functions, for example a horrible mess of unions and intersections of functions:

```
((number?) -> number?) | (((number) -> number) & ((string?) -> string?))
```

normalizes to an overloaded function:

```
((number) -> number?) & ((nil) -> (number | string)?)
```

Set-theoretic semantic subtyping does not support this normalization, and instead normalizes functions to *disjunctive normal form* (unions of intersections of functions). We do not do this for ergonomic reasons: overloaded functions are idiomatic in Luau, but DNF is not, and we do not want to present users with such non-idiomatic types.

Our normalization relies on rewriting away unions of function types:

```
((A) -> B) | ((C) -> D)   →   (A & C) -> (B | D)
```

This normalization is sound in our model, but not in set-theoretic models.

Secondly, in Luau, the type of a function application `f(x)` is `B` if `f` has type `(A) -> B` and `x` has type `A`. Unexpectedly, this is not always true in set-theoretic models, due to uninhabited types. In set-theoretic models, if `x` has type `never` then `f(x)` has type `never`. We do not want to burden users with the idea that function application has a special corner case, especially since that corner case can only arise in dead code.

In set-theoretic models, `(never) -> A` is a subtype of `(never) -> B`, no matter what `A` and `B` are. This is not true in Luau.

For these two reasons (which are largely about ergonomics rather than anything technical) we drop the set-theoretic requirement, and use *pragmatic* semantic subtyping.

Negation types

The other difference between Luau's type system and off-the-shelf semantic subtyping is that Luau does not support all negated types.

The common case for wanting negated types is in typechecking conditionals:

```
-- initially x has type T
if (type(x) == "string") then
  --  in this branch x has type T & string
else
  -- in this branch x has type T & ~string
end
```

This uses a negated type `~string` inhabited by values that are not strings.

In Luau, we only allow this kind of typing refinement on *test types* like `string`, `function`, `Part` and so on, and *not* on structural types like `(A) -> B`, which avoids the common case of general negated types.

Prototyping and verification

During the design of Luau's semantic subtyping algorithm, there were changes made (for example initially we thought we were going to be able to use set-theoretic subtyping). During this time of rapid change, it was important to be able to iterate quickly, so we initially implemented a prototype (https://github.com/luau-lang/agda-typeck) rather than jumping straight to a production implementation.

Validating the prototype was important, since subtyping algorithms can have unexpected corner cases. For this reason, we adopted Agda as the prototyping language. As well as supporting unit testing, Agda supports mechanized verification, so we are confident in the design.

The prototype does not implement all of Luau, just the functional subset, but this was enough to discover subtle feature interactions that would probably have surfaced as difficult-to-fix bugs in production.

Prototyping is not perfect, for example the main issues that we hit in production were about performance and the C++ standard library, which are never going to be caught by a prototype. But the production implementation was otherwise fairly straightforward (or at least as straightforward as a 3kLOC change can be).

Next steps

Semantic subtyping has removed one source of false positives, but we still have others to track down:

- overloaded function applications and operators,
- property access on expressions of complex type,
- read-only properties of tables,
- variables that change type over time (aka typestates),
- …

The quest to remove spurious red squiggles continues!

Acknowledgments

Thanks to Giuseppe Castagna and Ben Greenman for helpful comments on drafts of this post.

Further reading

If you want to find out more about Luau and semantic subtyping, you might want to check out…

- Luau. https://luau-lang.org/ (https://luau-lang.org/)
- Lily Brown, Andy Friesen and Alan Jeffrey, *Goals of the Luau Type System*, Human Aspects of Types and Reasoning Assistants (HATRA), 2021. https://arxiv.org/abs/2109.11397 (https://arxiv.org/abs/2109.11397)
- Luau Typechecker Prototype. https://github.com/luau-lang/agda-typeck (https://github.com/luau-lang/agda-typeck)
- Agda. https://agda.readthedocs.io/ (https://agda.readthedocs.io/)
- Andrew M. Kent. *Down and Dirty with Semantic Set-theoretic Types*, 2021. https://pnwamk.github.io/sst-tutorial/ (https://pnwamk.github.io/sst-tutorial/)
- Giuseppe Castagna, *Covariance and Contravariance*, Logical Methods in Computer Science 16(1), 2022. https://arxiv.org/abs/1809.01427 (https://arxiv.org/abs/1809.01427)
- Giuseppe Castagna and Alain Frisch, *A gentle introduction to semantic subtyping*, Proc. Principles and practice of declarative programming (PPDP), pp 198–208, 2005. https://doi.org/10.1145/1069774.1069793

(https://doi.org/10.1145/1069774.1069793)

- Giuseppe Castagna, Mickaël Laurent, Kim Nguyễn, Matthew Lutze, *On Type-Cases, Union Elimination, and Occurrence Typing*, Principles of Programming Languages (POPL), 2022. https://doi.org/10.1145/3498674 (https://doi.org/10.1145/3498674)
- Giuseppe Castagna, *Programming with union, intersection, and negation types*, 2022. https://arxiv.org/abs/2111.03354 (https://arxiv.org/abs/2111.03354)
- Sam Tobin-Hochstadt and Matthias Felleisen, *Logical types for untyped languages*. International Conference on Functional Programming (ICFP), 2010. https://doi.org/10.1145/1863543.1863561 (https://doi.org/10.1145/1863543.1863561)
- José Valim, *My Future with Elixir: set-theoretic types*, 2022. https://elixir-lang.org/blog/2022/10/05/my-future-with-elixir-set-theoretic-types/ (https://elixir-lang.org/blog/2022/10/05/my-future-with-elixir-set-theoretic-types/)

Some other languages which support semantic subtyping…

- ℂDuce https://www.cduce.org/ (https://www.cduce.org/)
- Ballerina https://ballerina.io (https://ballerina.io)
- Elixir https://elixir-lang.org/ (https://elixir-lang.org/)
- eqWAlizer https://github.com/WhatsApp/eqwalizer (https://github.com/WhatsApp/eqwalizer)

And if you want to see the production code, it's in the C++ definitions of tryUnifyNormalizedTypes (https://github.com/Roblox/luau/blob/d6aa35583e4be14304d2a17c7d11c8819756beb6/Analysis/src/Unifier.cpp#L868) and NormalizedType (https://github.com/Roblox/luau/blob/d6aa35583e4be14304d2a17c7d11c8819756beb6/Analysis/include/Luau/Normalize.h#L134) in the open source Luau repo (https://github.com/Roblox/luau).

---

layout: single title: "Luau Recap: September & October 2022"

Luau is our new language that you can read more about at https://luau-lang.org (https://luau-lang.org).

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-september-october-2022/).]

Semantic subtyping

One of the most important goals for Luau is to avoid *false positives*, that is cases where Script Analysis reports a type error, but in fact the code is correct. This is very frustrating, especially for beginners. Spending time chasing down a gnarly type error only to discover that it was the type system that's wrong is nobody's idea of fun!

We are pleased to announce that a major component of minimizing false positives has landed, *semantic subtyping*, which removes a class of false positives caused by failures of subtyping. For example, in the program

```
local x : CFrame = CFrame.new()
local y : Vector3 | CFrame
if (math.random()) then
  y = CFrame.new()
else
  y = Vector3.new()
end
local z : Vector3 | CFrame = x * y -- Type Error!
```

an error is reported, even though there is no problem at runtime. This is because `CFrame`'s multiplication has two overloads:

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

The current syntax-driven algorithm for subtyping is not sophisticated enough to realize that this is a subtype of the desired type:

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

Our new algorithm is driven by the semantics of subtyping, not the syntax of types, and eliminates this class of false positives.

If you want to know more about semantic subtyping in Luau, check out our technical blog post (https://luau-lang.org/2022/10/31/luau-semantic-subtyping.html) on the subject.

Other analysis improvements

- Improve stringification of function types.
- Improve parse error warnings in the case of missing tokens after a comma.
- Improve typechecking of expressions involving variadics such as `{ ... }`.
- Make sure modules don't return unbound generic types.
- Improve cycle detection in stringifying types.
- Improve type inference of combinations of intersections and generic functions.
- Improve typechecking when calling a function which returns a variadic e.g. `() -> (number...)`.
- Improve typechecking when passing a function expression as a parameter to a function.
- Improve error reporting locations.
- Remove some sources of memory corruption and crashes.

Other runtime and debugger improvements

- Improve performance of accessing debug info.
- Improve performance of `getmetatable` and `setmetatable`.
- Remove a source of freezes in the debugger.
- Improve GC accuracy and performance.

Thanks

Thanks for all the contributions!

- AllanJeremy (https://github.com/AllanJeremy)
- JohnnyMorganz (https://github.com/JohnnyMorganz)
- jujhar16 (https://github.com/jujhar16)
- petrihakkinen (https://github.com/petrihakkinen)

layout: single title: "Luau origins and evolution" author: Arseny Kapoulkine

At the heart of Roblox technology lies Luau (https://luau-lang.org), a scripting language derived from Lua 5.1 that is being developed (https://github.com/Roblox/luau) by an internal team of programming language experts with the help of open source contributors.

It powers all user-generated content on Roblox, providing access to a very rich set of APIs that allows manipulation of objects in the 3D world, backend API access, UI interaction and more. Hundreds of thousands of developers write code in Luau every month, with top experiences using hundreds of thousands of lines of code, adding up to hundreds of millions of lines of code across the platform. For many of them, it is the first programming language they learn, and one they spend the majority of their time programming in. Using a set of extended APIs developers also customize their workflows by writing plugins to Roblox Studio, where they work on their experiences, using an extended API surface to interact with all aspects of the editor.

It also powers a lot of application code that Roblox engineers are writing: Universal App, the gateway to the worlds of Roblox that is used by tens of millions of people every day, has 95% of its functionality implemented in Luau, and Roblox Studio has a lot of builtin critical functionality such as part and terrain editors, marketplace browser, avatar and animation editors, material manager and more, implemented in Luau as a plugin, mostly using the same APIs that developers have access to. Every week, updates to this internal codebase that is now over 2 million lines large, are shipped to all Roblox users.

In addition to Roblox use cases, Luau is also open-source and is seeing an increased adoption in other projects and applications.

But why did we use Lua in the first place, and why did we decide to pursue building a new language on top of it?

Early beginnings

Around 2006, when a very early version of the Roblox platform was developed, the question of user generated behaviors emerged. Before that, users were able to build non-interactive content on Roblox, and the only form of interaction was physics simulation. While this provided rich emergent behavior, it was hard to build gameplay on top of this: for example, to build a Capture The Flag game, you need to handle collision between players and flags spread throughout the map with a bit of logic that dictates how to adjust team points and when to remove or recreate the objects.

After an early and brief misstep when we decided to add a few gameplay objects to the core definition of Roblox worlds (some developers may recognize FlagStand as a class name...), the Roblox co-founder Erik Cassel realized that an approach like this is fundamentally limiting the power of user generated content. It’s not enough to give creators the basic blocks on top of which to build their creations, it’s critical to expose the power of a full Turing-complete programming language. Without this, the expressive capability and the reach of the platform would have been restricted far too much.

But which programming language to choose? This is where Lua (https://lua.org/), which was, and still is, one of the dominant programming languages used in video games, comes in.

In addition to its simplicity, which made the language easy to learn and get productive in, Lua was the fastest scripting language compared to popular alternatives like Python or JavaScript at the time[1], designed to be embedded which meant an easy ability to expose APIs from the host application to the scripts as well as high degree of execution control from the host, and implemented coroutines, a very powerful concurrency primitive that allowed to easily and intuitively script behaviors for independent actors in game using linear control flow.

Instead of having a large standard library, the expectation was that the embedding application would define a set of APIs that that application needed, as well as establish policies of running the code - which gave us a lot of freedom in how to structure the APIs and when the scripts would get triggered during the simulation of a single frame.

Power of simplicity

Lua is a simple language. What does simplicity mean for us?

Being a simple language means having a small set of features. Lua has all the fundamental features but doesn't have a lot of syntax sugar - this means the language is easier to teach and learn, and you rarely run into code that's difficult to understand syntactically because it uses an unfamiliar construct. Of course, this also means that some programs in Lua are longer than equivalent programs in languages that have more dedicated constructs to solve specific problems, such as list comprehensions in Python.

Being a simple language means having a minimal set of rules for every feature. Lua does deviate from this in certain respects (which is to say, the language could have been even simpler!), but notably for a dynamic language the behavior of fundamental operators is generally easy to explain and unsurprising - for example, two values in Lua are equal iff they have the same type and the same value, as such `0 == "0"` is `false`; as another example, `for` loops introduce unique variable bindings on every iteration, as such capturing the iteration variable in a closure produces unique values. These decisions lead to more concise and efficient implementation and eliminate a class of bugs in programs.

Being a simple language means having a small implementation. This may be immaterial to people writing code in the language, but it leads to an implementation that can be of higher quality; simpler implementations can also be easier to optimize for memory or performance, and are easier to build upon.

Developers on the Roblox platform have very diverse programming backgrounds. Some are writing their first line of code in Roblox Studio, while others have computer science degrees and experience working in multiple different programming languages. While it's always possible to support two different programming languages that target different segments of the audience, that fragments the ecosystem and makes the programming story less consistent (impacting documentation, tutorials, code reuse, ability for community members to help each other write code, presents challenges with interaction between different languages in the same experience and more). A better outcome is one where a single language can serve both audiences - this requires a language that strikes a balance between simplicity and generality, and while Lua isn't perfect here, it's great as a foundation for a language like this[2].

In many ways, Lua is simultaneously simple and pragmatic: many parts of the language are difficult to make much better without a lot of added complexity, but at the same time it requires little in the way of extra functionality to be able to solve problems efficiently. That said, no language is perfect, and within several areas of Lua we felt that the tradeoffs weren't quite right for our use case.

Respectful evolution

In 2019, we decided to build Luau (https://luau-lang.org) - a language derived from Lua and compatible with Lua 5.1, which is the version we've been using all these years. At the time we evaluated other routes, but ultimately settled on this as the most optimal long-term.

On one hand, we loved a lot of things about Lua - both design wise and implementation wise, while there were some decisions we felt were suboptimal, by and large it was an almost perfect foundation for what we've set out to achieve.

On the other hand, we've been running into the limitations of Lua on large code bases in absence of type checking, performance was good but not great, and some missing features would have been great to have.

Some of the things we've been missing have been added in later versions of Lua, yet we were still using Lua 5.1. While we would have loved to use a later version of the language standard, Lua 5.x releases are not backwards compatible, and some releases remove support for features that are in wide use at Roblox. For Roblox, backwards compatibility is an essential feature of the platform - while we don't have a guarantee that content created 10 years ago still works, to the extent that we can achieve that without restricting the platform evolution too much, we try.

What we've realized is that Lua is a great foundation for a perfect language that we can build for Roblox.

We would maintain backwards compatibility with Lua 5.1 but evolve the language from there; sometimes this means taking later features from Lua that don't conflict with the existing language or our design values, sometimes this means innovating beyond what Lua has done. Crucially, we must maintain the balance between simplicity and power - we still value simplicity, we still need to avoid a feature explosion to ensure that the features compose and are of high quality, and we still need the language to be a good fit for beginners.

One of the largest limitations that we've seen is the lack of type checking making it easy to make mistakes in large code bases, as such support for type checking (https://luau-lang.org/typecheck) was a requirement for Luau. However, it's important that the type checker is mostly transparent to the developers who don't want to invest the time to learn it - anything else would change the learning curve too much for the language to be suitable for beginners. As such, we've investing in gradual typing, and our type checker is learning to strike a balance between inferring useful types for completely untyped programs (which, among other things, greatly enhances editing experience through type-aware autocomplete), and the lack of false positive diagnostics that can be confusing and distracting.

While we did need to introduce extra syntax (https://luau-lang.org/syntax) to the language - most notably, to support optional type annotations - it was important for us to maintain the cohesion of the overall syntax. We aren't seeking to make a new language with a syntax alien to Lua programmers - Luau programs are still recognizably Lua, and to the extent possible we try to avoid new syntactic features. In a sense, we still want the syntax, semantics, and the runtime to be simple and minimal - but at the same time we have important problems to solve with respect to ergonomics, robustness and performance of the language, and solving some of them requires having slightly more complex syntax, semantics, or implementation.

So in finding ways to evolve Luau, we strive to design features that feel like they would be at home in Lua. At the same time, we've adopted a more open evolution process - the language development is driven through RFCs (https://github.com/Roblox/luau/blob/master/rfcs/README.md) that are designs open to the public that anyone can contribute to - this is in contrast with Lua, which has a very closed development process, and is one of the reasons why it would have been difficult for us to keep using Lua as we wouldn't get a say in its development. At the same time, to ensure the design criterias are met, it's important that the Luau development team at Roblox maintains a final say over design and implementation of the language[3], while taking the community's proposals and input into consideration.

Importance of co-design

Luau language is developed in concert with the language compiler, runtime, type checker and other analysis tools, autocomplete engine and other tooling, and that development is guided by the vast volume of existing Luau code, both internal and external.

This is one of the key principles behind our evolution philosophy - neither layer is developed in isolation, and instead concerns at every level inform all other aspects of the language design and implementation.

This means that when designing language features, we make sure that they can be implemented efficiently, type checked properly, can be supported well in editing and analysis tools and have a positive impact on the code internal and external engineers write. When we find issues in any component, we can always ask, what changes to other components or even language design would make for a better overall solution.

This avoids some classes of design problems, for example we won't specify a language feature that has a prohibitively high implementation cost, as it violates our simplicity criteria, or that is impractical to implement efficiently, as that would create a performance hazard. This also means that when implementing various components of the language we cross-check the concerns and applicability of these across the entire stack - for example, we've reworked our auto-complete system to use the same type inference engine that the type checking / analysis tools use, which had immense benefits for the experience of editing code, but also applied significant back pressure on the type inference itself, forcing us to improve it substantially and fix a lot of corner cases that would otherwise have lingered unnoticed.

Whenever we develop features, optimizations, improve our analysis engine or enhance the standard libraries, we also heavily rely on code written in Luau to validate our hypotheses. When working on new features we find motivation in the real problems that we see our developers face. For example, we implemented the new ternary operator (https://luau-lang.org/syntax#if-then-else-expressions) after seeing a large set of cases where existing Lua's `a and b or c` pattern was error-prone for boolean values, which made it easy to accidentally introduce a mistake that was hard to identify automatically. All optimizations and new analysis features are validated on our internal 2M LOC codebase before being added to Luau, which allows us to quickly get initial validation of ideas, or invalidate some approaches as infeasible / unhelpful.

In addition to that, while we don't have direct access to community-developed source code for privacy reasons, we can run experiments and collect telemetry[4], which also helps us make decisions regarding backwards compatibility. Due to Hyrum's law (https://www.hyrumslaw.com/), technically any change in the language or libraries, no matter how small, would be backwards incompatible - instead we adopt the notion of pragmatic balance between strict backwards compatibility[5] and pragmatic compatibility concerns. For example, later versions of Lua make some library functions like `table.insert`/`table.remove` more strict with how they handle out of range indices. We have evaluated this change for compatibility by collecting telemetry on the use of out of range indices in these functions on the Roblox platform and concluded that applying the stricter checking would break existing programs, and instead had to slightly adjust the rules for out of range behavior in ways that was benign for existing code but prevented catastrophic performance degradation for large out of range indices. Because we couldn't afford to introduce new runtime errors in this case, we also added a set of linting rules to our analysis engine to flag potential misuse of `table.insert`/`table.remove` before the code ever gets to run - this diagnostics is informational and as such doesn't affect backwards compatibility, but does help prevent mistakes.

There are also cases where this co-design approach prevents introduction of features that can lead to easy misuse, which can be difficult to see in the design of the feature itself, but becomes more apparent when you consider features in context of the entire ecosystem. This is a good thing - it means co-design acts as a forcing function on the language simplicity and makes it easier to flag potential bad interactions between different language features, or language features and tooling, or language features and existing programming patterns that are in widespread use in real-world code. By making sure that all features are validated for their impact across the stack and on code written in Luau, we ultimately get a better, simpler and more cohesive language.

Efficient execution

One of the critical goals in front of Luau is efficiency, both from the performance and memory perspective. There's only so many milliseconds in a frame, and we simultaneously see the need to increase the scale and complexity of simulated experiences, which requires more memory and computation, as well as the need to fit more comfortably into smaller budgets of performance memory for better experience on smaller devices. In fact, one of the motivations for Luau in 2019 has been improved performance, as we saw many opportunities to go beyond Lua with a redesigned implementation.

Crucially, our performance needs are somewhat unique and require somewhat unique solutions.

We need Luau to run on many platforms where native code generation is either prohibited by the platform vendor or impractical due to tight memory constraints. As such, in terms of execution performance it's critical that we have a very fast interpreter[6]. However, we have freedom in terms of high level design of the entire stack - for example, clients never see the source code of the scripts as all compilation to bytecode happens on the server; this gives us an opportunity to perform more involved and expensive optimizations during that process as well as have the smallest possible startup time on the client without complex pre-parse steps. Notably, our bytecode compiler performs a series of high level optimizations including function inlining and loop unrolling that in other dynamic languages is often left to the just-in-time compiler.

Another area where performance is critical is garbage collection. Garbage collection is crucial for the language's simplicity as it makes memory management easier to reason about, but it does require a substantial amount of implementation effort to keep it efficient. For Roblox and for any other game engine or interactive simulation, latency is critical and so our collector is heavily optimized for that - to the extent possible collection is incremental and stop-the-world pauses are very brief. Another part of the performance story here however is the language and data structure design - by making sure that core data types are efficient in how they are laid out in memory we reduce the amount of work garbage collector takes to trace the heap, and, as another example of co-design, we try to make sure that language features are conscious of the impact they have on memory and garbage collection efficiency.

However, from a whole-platform standpoint there's a lot of performance aspects that go beyond single-threaded execution. This is an active area of research and development for the team, as to really leverage the hardware the code is running on we need to think about SIMD, hardware thread utilization as well as running code in a cluster of nodes. These considerations inform current and future development of the runtime and the language (for example, our runtime now supports efficient operations on short SIMD vectors even in interpreted mode, and the VM is fairly lightweight to instantiate which makes running many VMs per core practical, with message passing or access to shared Roblox data model used to make gameplay features feasible to implement), but we're definitely in the early days here - our first implementation of parallel script execution in Roblox just shipped earlier this year (https://devforum.roblox.com/t/full-release-of-parallel-luau-v1/1836187). This is likely the area where a lot of future innovations will happen as well.

Future

We're very happy with the success of Luau - in several years we've established consistent processes for evolving the language and so far we found a good balance between simplicity, ease of use, performance and robustness of the language, its implementation and the tooling surrounding it. The language keeps continuously evolving but at a pace that is easy to stay on top of - in 2022 we shipped a few syntactic extensions for type annotations but no changes to the syntax of the language outside of types, and only one major semantic change to the for loop iteration (https://luau-lang.org/syntax#generalized-iteration) that actually made the language easier to use by avoiding the need to specify the table traversal style via `pairs`/`ipairs`. We try to make sure that the features are general and provide enough extensibility so that libraries can be built on top of the language to make it easier to write code, while also making it practical to use the language without complex supporting frameworks.

There's still a lot of ground to cover, and we'll be working on Luau for years to come. We're in the process of building the next version of our type inference / checking engine to make sure that all users of the language regardless of their expertise benefit from it, we've started investing in native code generation as we're reaching the limits of interpreted performance (although some exciting opportunities for compiler optimization are still on the horizon), and there's still a lot of hard design and implementation work ahead of us for some important language features and standard libraries. And as mentioned, our execution model will likely see a lot of innovation as we push the boundaries of hardware utilization across cores and nodes.

Overall, Luau is like an iceberg - the surface is simple to learn and use, but it hides the tremendous amount of careful design, engineering and attention to detail, and we plan to continue to invest in it while trying to keep the outer surface comparatively small. We're excited to see how far we can take it!

1. High-performance JavaScript engines didn’t exist at the time! LuaJIT was around the corner and redefined the performance expectations of dynamic languages.↩

2. In fact, scaling to large teams of expert programmers is one of the core motivations behind our creating Luau, while a requirement to still be suitable for beginner programmers guides our evolution direction.↩

3. This would have been difficult to drive in any existing large established language like JavaScript or Python.↩

4. This is limited to Roblox platform and doesn't exist in open-source releases.↩

5. Which we do follow in some areas, such as syntactic compatibility - all existing programs that parse must continue to parse the same way as the language evolves.↩

6. Some design decisions and implementation techniques are documented on our performance page (https://luau-lang.org/performance).↩

---

layout: single title: "String Interpolation"

String interpolation is the new syntax introduced to Luau that allows you to create a string literal with expressions inside of that string literal.

In short, it's a safer and more ergonomic alternative over `string.format`.

Here's a quick example of a string interpolation:

```
local combos = {2, 7, 1, 8, 5}
print(`The lock combination is {table.concat(combos)}. Again, {table.concat(combos, ", ")}.`)
--> The lock combination is 27185. Again, 2, 7, 1, 8, 5.
```

String interpolation also composes well with the `__tostring` metamethod.

```
local balance = setmetatable({ value = 500 }, {
    __tostring = function(self)
        return "$" .. tostring(self.value)
    end
})

print(`You have {balance}!`)
--> You have $500!
```

To find out more details about this feature, check out Luau Syntax page (/syntax#string-interpolation).

This is also the first major language feature implemented in a contribution (https://github.com/Roblox/luau/pull/614) from the open-source community. Thanks Kampfkarren (https://github.com/Kampfkarren)!

layout: single title: "Luau Recap: March 2023"

How the time flies! The team has been busy since the last November Luau Recap working on some large updates that are coming in the future, but before those arrive, we have some improvements that you can already use!

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-march-2023/).]

Improved type refinements

Type refinements handle constraints placed on variables inside conditional blocks.

In the following example, while variable `a` is declared to have type `number?`, inside the `if` block we know that it cannot be `nil`:

```
local function f(a: number?)
    if a ~= nil then
        a *= 2 -- no type errors
    end
    ...
end
```

One limitation we had previously is that after a conditional block, refinements were discarded.

But there are cases where `if` is used to exit the function early, making the following code essentially act as a hidden `else` block.

We now correctly preserve such refinements and you should be able to remove `assert` function calls that were only used to get rid of false positive errors about types being `nil`.

```
local function f(x: string?)
    if not x then return end

    -- x is a 'string' here
end
```

Throwing calls like `error()` or `assert(false)` instead of a `return` statement are also recognized.

```
local function f(x: string?)
    if not x then error('first argument is nil') end

    -- x is 'string' here
end
```

Existing complex refinements like `type`/`typeof`, tagged union checks and other are expected to work as expected.

Marking table.getn/foreach/foreachi as deprecated

`table.getn`, `table.foreach` and `table.foreachi` were deprecated in Lua 5.1 that Luau is based on, and removed in Lua 5.2.

`table.getn(x)` is equivalent to `rawlen(x)` when 'x' is a table; when 'x' is not a table, `table.getn` produces an error.

It's difficult to imagine code where `table.getn(x)` is better than either `#x` (idiomatic) or `rawlen(x)` (fully compatible replacement).

`table.getn` is also slower than both alternatives and was marked as deprecated.

`table.foreach` is equivalent to a `for .. pairs` loop; `table.foreachi` is equivalent to a `for .. ipairs` loop; both may also be replaced by generalized iteration.

Both functions are significantly slower than equivalent for loop replacements, are more restrictive because the function can't yield.

Because both functions bring no value over other library or language alternatives, they were marked deprecated as well.

You may have noticed linter warnings about places where these functions are used. For compatibility, these functions are not going to be removed.

Autocomplete improvements

When table key type is defined to be a union of string singletons, those keys can now autocomplete in locations marked as '^':

```
type Direction = "north" | "south" | "east" | "west"

local a: {[Direction]: boolean} = {[^] = true}
local b: {[Direction]: boolean} = {["^"]}
local b: {[Direction]: boolean} = {^}
```

We also fixed incorrect and incomplete suggestions inside the header of `if`, `for` and `while` statements.

Runtime improvements

On the runtime side, we added multiple optimizations.

`table.sort` is now ~4.1x faster (when not using a predicate) and ~2.1x faster when using a simple predicate.

We also have ideas on how improve the sorting performance in the future.

`math.floor`, `math.ceil` and `math.round` now use specialized processor instructions. We have measured ~7-9% speedup in math benchmarks that heavily used those functions.

A small improvement was made to builtin library function calls, getting a 1-2% improvement in code that contains a lot of fastcalls.

Finally, a fix was made to table array part resizing that brings large improvement to performance of large tables filled as an array, but at an offset (for example, starting at 10000 instead of 1).

Aside from performance, a correctness issue was fixed in multi-assignment expressions.

```
arr[1], n = n, n - 1
```

In this example, `n - 1` was assigned to `n` before `n` was assigned to `arr[1]`. This issue has now been fixed.

Analysis improvements

Multiple changes were made to improve error messages and type presentation.

- Table type strings are now shown with newlines, to make them easier to read
- Fixed unions of `nil` types displaying as a single `?` character
- "Type pack A cannot be converted to B" error is not reported instead of a cryptic "Failed to unify type packs"
- Improved error message for value count mismatch in assignments like `local a, b = 2`

You may have seen error messages like `Type 'string' cannot be converted to 'string?'` even though usually it is valid to assign `local s: string? = 'hello'` because `string` is a sub-type of `string?`.

This is true in what is called Covariant use contexts, but doesn't hold in Invariant use contexts, like in the example below:

```
local a: { x: Model }
local b: { x: Instance } = a -- Type 'Model' could not be converted into 'Instance' in an invariant context
```

In this example, while `Model` is a sub-type of `Instance` and can be used where `Instance` is required.

The same is not true for a table field because when using table `b`, `b.x` can be assigned an `Instance` that is not a `Model`. When `b` is an alias to `a`, this assignment is not compatible with `a`'s type annotation.

Some other light changes to type inference include:

- `string.match` and `string.gmatch` are now defined to return optional values as match is not guaranteed at runtime
- Added an error when unrelated types are compared with `==`/`~=`
- Fixed issues where variable after `typeof(x) == 'table'` could not have been used as a table

Thanks

A very special thanks to all of our open source contributors:

- niansa/tuxifan (https://github.com/niansa)
- B. Gibbons (https://github.com/bmg817)
- Epix (https://github.com/EpixScripts)
- Harold Cindy (https://github.com/HaroldCindy)
- Qualadore (https://github.com/Qualadore)

layout: single title: "Luau Recap: July 2023"

Our team is still spending a lot of time working on upcoming replacement for our type inference engine as well as working on native code generation to improve runtime performance.

However, we also worked on unrelated improvements during this time that are summarized here.

[Cross-posted to the Roblox Developer Forum (https://devforum.roblox.com/t/luau-recap-july-2023/).]

Analysis improvements

Indexing table intersections using `x["prop"]` syntax has been fixed and no longer reports a false positive error:

```
type T = { foo: string } & { bar: number }
local x: T = { foo = "1", bar = 2 }

local y = x["bar"] -- This is no longer an error
```

Generic `T...` type is now convertible to `...any` variadic parameter.

This solves issues people had with variadic functions and variadic argument:

```
local function foo(...: any)
    print(...)
end

local function bar<T...>(...: T...)
    foo(...) -- This is no longer an error
end
```

We have also improved our general typechecking performance by ~17% and by additional ~30% in modules with complex types.

Other fixes include:

- Fixed issue with type `T?` not being convertible to `T | T` or `T?` which could've generated confusing errors
- Return type of `os.date` is now inferred as `DateTypeResult` when argument is "*t" or "!*t"

Runtime improvements

Out-of-memory exception handling has been improved. `xpcall` handlers will now actually be called with a "not enough memory" string and whatever string/object they return will be correctly propagated.

Other runtime improvements we've made:

- Performance of `table.sort` was improved further. It now guarantees N*log(N) time complexity in the worst case
- Performance of `table.concat` was improved by ~5-7%
- Performance of `math.noise` was improved by ~30%
- Inlining of functions is now possible even when they used to compute their own arguments
- Improved logic for determining whether inlining a function or unrolling a loop is profitable

Autocomplete improvements

An issue with exported types not being suggested is now fixed.

Debugger improvements

We have fixed the search for the closest executable breakpoint line.

Previously, breakpoints might have been skipped in `else` blocks at the end of a function. This simplified example shows the issue:

```
local function foo(isIt)
    if isIt then
        print("yes")
    else
        -- When 'true' block exits the function, breakpoint couldn't be placed here
        print("no")
    end
end
```

Thanks

A very special thanks to all of our open source contributors:

- Petri Häkkinen (https://github.com/petrihakkinen)
- JohnnyMorganz (https://github.com/JohnnyMorganz)
- Gael (https://github.com/TheGreatSageEqualToHeaven)
- Jan (https://github.com/Jan200101)
- Alex Orlenko (https://github.com/khvzak)
- mundusnine (https://github.com/mundusnine)
- Ben Mactavsin (https://github.com/BenMactavsin)

- RadiatedExodus (https://github.com/RealEthanPlayzDev)
- Lodinu Kalugalage (https://github.com/imlodinu)
- MagelessMayhem (https://github.com/MagelessMayhem)
- Someon1e (https://github.com/Someon1e)

---

layout: single title: "Luau Recap: October 2023"

We're still quite busy working on some big type checking updates that we hope to talk about soon, but we have a few equally exciting updates to share in the meantime!

Let's dive in!

Floor Division

Luau now has a floor division operator. It is spelled `//`:

```
local a = 10 // 3 -- a == 3
a //= 2           -- a == 1
```

For numbers, `a // b` is equivalent to `math.floor(a / b)`, and you can also overload this operator by implementing the `__idiv` metamethod. The syntax and semantics are borrowed from Lua 5.3 (although Lua 5.3 has an integer type while we don't, we tried to match the behavior to be as close as possible).

Native Codegen Preview

We are actively working on our new native code generation module that can significantly improve the performance of compute-dense scripts by compiling them to X64 (Intel/AMD) or A64 (ARM) machine code and executing that natively. We aim to support all AArch64 hardware with the current focus being Apple Silicon (M1-M3) chips, and all Intel/AMD hardware that supports AVX1 (with no planned support for earlier systems). When the hardware does not support native code generation, any code that would be compiled as native just falls back to the interpreted execution.

When working with open-source releases (https://github.com/luau-lang/luau/releases), binaries now have native code generation support compiled in by default; you need to pass `--codegen` command line flag to enable it. If you use Luau as a library in a third-party application, you would need to manually link `Luau.CodeGen` library and call the necessary functions to compile specific modules as needed - or keep using the interpreter if you want to! If you work in Roblox Studio, we have integrated native code generation preview as a beta feature (https://devforum.roblox.com/t/luau-native-code-generation-preview-studio-beta/2572587), which currently requires manual annotation of select scripts with `--!native` comment.

Our goal for the native code generation is to help reach ultimate performance for code that needs to process data very efficiently, but not necessarily to accelerate every line of code, and not to replace the interpreter. We remain committed to maximizing interpreted execution performance, as not all platforms will support native code generation, and it's not always practical to use native code generation for large code bases because it has a larger memory impact than bytecode. We intend for this to unlock new performance opportunities for complex features and algorithms, e.g. code that spends a lot of time working with numbers and arrays, but not to dramatically change performance on UI code or code that spends a lot of its time calling Lua functions like `table.sort`, or external C functions (like Roblox engine APIs).

Importantly, native code generation does not change our behavior or correctness expectations. Code compiled natively should give the same results when it executes as non-native code (just take a little less time), and it should not result in any memory safety or sandboxing issues. If you ever notice native code giving a different result from non-native code, please submit a bug report.

We continue to work on many code size and performance improvements; here's a short summary of what we've done in the last couple of months, and there's more to come!

- Repeated access to table fields with the same object and name are now optimized (e.g. `t.x = t.x + 5` is faster)
- Numerical `for` loops are now compiled more efficiently, yielding significant speedups on hot loops
- Bit operations with constants are now compiled more efficiently on X64 (for example, `bit32.lshift(x, 1)` is faster); this optimization was already in place for A64
- Repeated access to array elements with the same object and index is now faster in certain cases
- Performance of function calls has been marginally improved on X64 and A64
- Fix code generation for some `bit32.extract` variants where we could produce incorrect results
- `table.insert` is now faster when called with two arguments as it's compiled directly to native code
- To reduce code size, module code outside of functions is not compiled natively unless it has loops

Analysis Improvements

The `break` and `continue` keywords can now be used in loop bodies to refine variables. This was contributed by a community member - thank you, AmberGraceSoftware (https://github.com/AmberGraceSoftware)!

```
function f(objects: { { value: string? } })
    for _, object in objects do
        if not object.value then
            continue
        end

        local x: string = object.value -- ok!
    end
end
```

When type information is present, we will now emit a warning when `#` or `ipairs` is used on a table that has no numeric keys or indexers. This helps avoid common bugs like using `#t == 0` to check if a dictionary is empty.

```
local message = { data = { 1, 2, 3 } }

if #message == 0 then -- Using '#' on a table without an array part is likely a bug
end
```

Finally, some uses of `getfenv`/`setfenv` are now flagged as deprecated. We do not plan to remove support for `getfenv`/`setfenv` but we actively discourage its use as it disables many optimizations throughout the compiler, runtime, and native code generation, and interferes with type checking and linting.

Autocomplete Improvements

We used to have a bug that would arise in the following situation:

```
--!strict
type Direction = "Left" | "Right"
local dir: Direction = "Left"

if dir == ""| then
end
```

(imagine the cursor is at the position of the `|` character in the `if` statement)

We used to suggest `Left` and `Right` even though they are not valid completions at that position. This is now fixed.

We've also added a complete suggestion for anonymous functions if one would be valid at the requested position. For example:

```
local p = Instance.new('Part')
p.Touched:Connect(
```

You will see a completion suggestion `function (anonymous autofilled)`. Selecting that will cause the following to be inserted into your code:

```
local p = Instance.new('Part')
p.Touched:Connect(function(otherPart: BasePart)  end
```

We also fixed some confusing editor feedback in the following case:

```
game:FindFirstChild(
```

Previously, the signature help tooltip would erroneously tell you that you needed to pass a `self` argument. We now correctly offer the signature `FindFirstChild(name: string, recursive: boolean?): Instance`

Runtime Improvements

- `string.format`'s handling of `%*` and `%s` is now 1.5-2x faster
- `tonumber` and `tostring` are now 1.5x and 2.5x faster respectively when working on primitive types
- Compiler now recognizes `math.pi` and `math.huge` and performs constant folding on the expressions that involve these at `-O2`; for example, `math.pi*2` is now free.
- Compiler now optimizes `if...then...else` expressions into AND/OR form when possible (for example, `if x then x else y` now compiles as `x or y`)
- We had a few bugs around `repeat..until` statements when the `until` condition referred to local variables defined in the loop body. These bugs have been fixed.
- Fix an oversight that could lead to `string.char` and `string.sub` generating potentially unlimited amounts of garbage and exhausting all available memory.
- We had a bug that could cause the compiler to unroll loops that it really shouldn't. This could result in massive bytecode bloat. It is now fixed.

luau-lang on GitHub

If you've been paying attention to our GitHub projects, you may have noticed that we've moved `luau` repository to a new luau-lang GitHub organization (https://github.com/luau-lang)! This is purely an organizational change but it's helping us split a few repositories for working with documentation and RFCs and be more organized with pull requests in different areas. Make sure to update your bookmarks and star our main repository (https://github.com/luau-lang/luau) if you haven't already!

Lastly, a big thanks to our open source community (https://github.com/luau-lang/luau) for their generous contributions:

- MagelessMayhem (https://github.com/MagelessMayhem)
- cassanof (https://github.com/cassanof)
- LoganDark (https://github.com/LoganDark)
- j-hui (https://github.com/j-hui)
- xgqt (https://github.com/xgqt)
- jdpatdiscord (https://github.com/jdpatdiscord)
- Someon1e (https://github.com/Someon1e)
- AmberGraceSoftware (https://github.com/AmberGraceSoftware)
- RadiantUwU (https://github.com/RadiantUwU)
- SamuraiCrow (https://github.com/SamuraiCrow)

permalink: /compatibility title: Compatibility toc: true

Luau is based on Lua 5.1, and as such incorporates all features of 5.1, except for ones that had to be taken out due to sandboxing limitations. Because of backwards compatibility constraints, we don't remove features deprecated by later versions (e.g. we still support `getfenv`/`setfenv`). Later Lua versions introduce new features into the language and new libraries/functions.

Our overall goal is to incorporate features from the later versions of Lua when it makes sense for us to do so - the motivations behind some newer features are unclear or don't apply to the domain Luau is used in, and many features carry costs that don't always make sense to pay. The rest of this document describes the status of all features of Lua 5.2 and beyond, with the following classification:

- ✔ - the feature is available in Luau
- □ - the feature is not available in Luau because we don't believe it makes sense to include it
- 😞 - the feature is not available in Luau because of compatibility/sandboxing concerns
- □ - the feature is not available in Luau yet but we'd like to include it and are possibly working on it
- □♀ - the feature is not available in Luau yet; we don't have strong opinions on it so it might make it at some point

Please note that all of these decisions are not final, they are just our current stance. In some cases evolution of our VM may make a feature that was previously impractical to support due to performance complications feasible. In some cases a feature that didn't have a strong use case gains one, so we can implement it.

Implementation limits

Luau has certain limitations around the number of local variables, registers, upvalues, constants and instructions. These limits are often different from the limits imposed by various versions of Lua, and are documented here without promising that future versions will adhere to these. Note that writing code that is close to any of these limits is dangerous because this code may become invalid as our codegen evolves.

- Local variables: 200 per function (same as all versions of Lua, this includes function arguments)
- Upvalues: 200 per function (up from 60 in Lua 5.1)
- Registers: 255 per function (same as all versions of Lua, this includes local variables and function arguments)
- Constants: $2^{23\text{ per function (up from }2}$18 in Lua 5.1)
- Instructions: $2^{23\text{ per function (up from }2}$17 in Lua 5.1, although in both cases the limit only applies to control flow)
- Nested functions: $2^{15\text{ per function (down from }2}$18 in Lua 5.1)
- Stack depth: 20000 Lua calls per Lua thread, 200 C calls per C thread (e.g. `coroutine.resume`/`pcall` nesting is limited to 200)

Note that Lua 5.3 has a larger upvalue limit (255) and a larger constant limit (2^26); existing Luau limits are likely sufficient for reasonable use cases.

Lua 5.1

Since several features were removed from Lua 5.1 for sandboxing reasons, this table lists them for completeness.

feature	notes
`io`, `os`, `package` and `debug` library	note that some functions in `os`/`debug` are still present
`loadfile`, `dofile`	removed for sandboxing, no direct file access
`loadstring` bytecode and `string.dump`	exposing bytecode is dangerous for sandboxing reasons
`newproxy` can only be called with nil or boolean	extra flexibility removed for sandboxing

Sandboxing challenges are covered in the dedicated section (sandbox).

Lua 5.2

feature	status	notes
yieldable pcall/xpcall	✔	
yieldable metamethods	□	significant performance implications
ephemeron tables	□	this complicates and slows down the garbage collector esp. for large weak tables
emergency garbage collector	□	Luau runs in environments where handling memory exhaustion in emergency situations is not tenable
goto statement	□	this complicates the compiler, makes control flow unstructured and doesn't address a significant need
finalizers for tables	□	no `__gc` support due to sandboxing and performance/complexity
no more fenv for threads or functions	😟	we love this, but it breaks compatibility
tables honor the `__len` metamethod	✔	
hex and `\z` escapes in strings	✔	
support for hexadecimal floats	□♀	no strong use cases
order metamethods (`__lt`/`__le`) are called for unrelated metatables	□	no strong use cases and more complicated semantics, compatibility and performance implications
empty statement	□♀	less useful in Lua than in JS/C#/C/C++
`break` statement may appear in the middle of a block	□♀	we'd like to do it consistently for `break`/`return`/`continue` but there be dragons
arguments for function called through `xpcall`	✔	
optional base in `math.log`	✔	
optional separator in `string.rep`	□♀	no strong use cases

feature	status	notes
new metamethods `__pairs` and `__ipairs`	☐	superseded by `__iter`
frontier patterns	✔	
`%g` in patterns	✔	
`\0` in patterns	✔	
`bit32` library	✔	
`string.gsub` is stricter about using `%` on special characters only	✔	
light C functions	😞	this changes semantics of fenv on C functions and has complex implications wrt runtime performance
NaN keys are supported for tables with `__newindex`	✔	

Two things that are important to call out here are various new metamethods for tables and yielding in metamethods. In both cases, there are performance implications to supporting this - our implementation is *very* highly tuned for performance, so any changes that affect the core fundamentals of how Lua works have a price. To support yielding in metamethods we'd need to make the core of the VM more involved, since almost every single "interesting" opcode would need to learn how to be resumable - which also complicates future JIT/AOT story. Metamethods in general are important for extensibility, but very challenging to deal with in implementation, so we err on the side of not supporting any new metamethods unless a strong need arises.

For `__pairs`/`__ipairs`, we felt that extending library functions to enable custom containers wasn't the right choice. Instead we revisited iteration design to allow for self-iterating objects via `__iter` metamethod, which results in a cleaner iteration design that also makes it easier to iterate over tables. As such, we have no plans to support `__pairs`/`__ipairs` as all use cases for it can now be solved by `__iter`.

Ephemeron tables may be implemented at some point since they do have valid uses and they make weak tables semantically cleaner, however the cleanup mechanism for these is expensive and complicated, and as such this can only be considered after the pending GC rework is complete.

Lua 5.3

feature	status	notes
`\u` escapes in strings	✔	
integers (64-bit by default)	☐	backwards compatibility and performance implications
bitwise operators	☐	`bit32` library covers this in absence of 64-bit integers
basic utf-8 support	✔	we include `utf8` library and other UTF8 features
functions for packing and unpacking values (string.pack/unpack/packsize)	✔	
floor division	✔	

feature	status	notes
`ipairs` and the `table` library respect metamethods	□	no strong use cases, performance implications
new function `table.move`	✔	
`collectgarbage("count")` now returns only one result	✔	
`coroutine.isyieldable`	✔	
stricter error checking for `table.insert`/`table.remove`	😟	we love this, but it breaks compatibility
`__eq` metamethod is called for unrelated metatables	□	backwards compatibility and typechecking implications

It's important to highlight integer support and bitwise operators. For Luau, it's rare that a full 64-bit integer type is necessary - double-precision types support integers up to 2^53 (in Lua which is used in embedded space, integers may be more appealing in environments without a native 64-bit FPU). However, there's a *lot* of value in having a single number type, both from performance perspective and for consistency. Notably, Lua doesn't handle integer overflow properly, so using integers also carries compatibility implications.

If integers are taken out of the equation, bitwise operators make less sense, as integers aren't a first class feature; additionally, `bit32` library is more fully featured (includes commonly used operations such as rotates and arithmetic shift; bit extraction/replacement is also more readable). Adding operators along with metamethods for all of them increases complexity, which means this feature isn't worth it on the balance. Common arguments for this include a more familiar syntax, which, while true, gets more nuanced as `^` isn't available as a xor operator, and arithmetic right shift isn't expressible without yet another operator, and performance, which in Luau is substantially better than in Lua because `bit32` library uses VM builtins instead of expensive function calls.

Lua 5.4

feature	status	notes
new generational mode for garbage collection	□	we're working on gc optimizations and generational mode is on our radar
to-be-closed variables	□	the syntax is inconsistent with how we'd like to do attributes long-term; no strong use cases in our domain
const variables	□	while there's some demand for const variables, we'd never adopt this syntax
new implementation for math.random	✔	our RNG is based on PCG, unlike Lua 5.4 which uses Xoroshiro
optional `init` argument to `string.gmatch`	□♀	no strong use cases
new functions `lua_resetthread` and `coroutine.close`	✔	
coercions string-to-number moved to the string library	😟	we love this, but it breaks compatibility
new format `%p` in `string.format`	□♀	no strong use cases
`utf8` library accepts codepoints up to 2^31	□♀	no strong use cases

feature	status	notes
The use of the `__lt` metamethod to emulate `__le` has been removed	□	breaks compatibility and complicates comparison overloading story
When finalizing objects, Lua will call `__gc` metamethods that are not functions	□	no `__gc` support due to sandboxing and performance/complexity
The function print calls `__tostring` instead of tostring to format its arguments.	✔	
By default, the decoding functions in the utf8 library do not accept surrogates.	😟	breaks compatibility and doesn't seem very interesting otherwise

Taking syntax aside (which doesn't feel idiomatic or beautiful), `<close>` isn't very useful in Luau - its dominant use case is for code that works with external resources like files or sockets, but we don't provide such APIs - and has a very large complexity cost, evidences by a lot of bug fixes since the initial implementation in 5.4 work versions. `<const>` in Luau doesn't matter for performance - our multi-pass compiler is already able to analyze the usage of the variable to know if it's modified or not and extract all performance gains from it - so the only use here is for code readability, where the `<const>` syntax is... suboptimal.

If we do end up introducing const variables, it would be through a `const var = value` syntax, which is backwards compatible through a context-sensitive keyword similar to `type`. That said, there's ambiguity wrt whether `const` should simply behave like a read-only variable, ala JavaScript, or if it should represent a stronger contract, for example by limiting the expressions on the right hand side to ones compiler can evaluate ahead of time, or by freezing table values and thus guaranteeing immutability.

Differences from Lua

We have a few behavior deviations from Lua 5.x that come from either a different implementation, or our desire to clean up small inconsistencies in the language/libraries:

- Tail calls are not supported to simplify implementation, make debugging/stack traces more predictable and allow deep validation of caller identity for security
- Order of table assignment in table literals follows program order in mixed tables (Lua 5.x assigns array elements first in some cases)
- Equality comparisons call `__eq` metamethod even when objects are rawequal (which matches other metamethods like <= and facilitates NaN checking)
- `function()` expressions may reuse a previously created closure in certain scenarios (when all upvalues captured are the same) for efficiency, which changes object identity but doesn't change call semantics -- this is different from Lua 5.1 but similar to Lua 5.2/5.3
- `os.time` returns UTC timestamp when called with a table for consistency

---

permalink: /demo title: Demo classes: wide

{% include repl.html %}

---

permalink: /getting-started title: Getting Started toc: true

To get started with Luau you need to use `luau` command line binary to run your code and `luau-analyze` to run static analysis (including type checking and linting). You can download these from a recent release (https://github.com/luau-lang/luau/releases).

Creating a script

To create your own testing script, create a new file with `.luau` as the extension:

```
function ispositive(x)
    return x > 0
end

print(ispositive(1))
print(ispositive("2"))

function isfoo(a)
    return a == "foo"
end

print(isfoo("bar"))
print(isfoo(1))
```

You can now run the file using `luau test.luau` and analyze it using `luau-analyze test.luau`.

Note that there are no warnings about calling `ispositive()` with a string, or calling `isfoo()` a number. This is because the type checking uses non-strict mode by default, which is lenient in how it infers types used by the program.

Type inference

Now modify the script to include `--!strict` at the top:

```
--!strict

function ispositive(x)
    return x > 0
end

print(ispositive(1))
print(ispositive("2"))
```

In `strict` mode, Luau will infer types based on analysis of the code flow. There is also `nonstrict` mode, where analysis is more conservative and types are more frequently inferred as `any` to reduce cases where legitimate code is flagged with warnings.

In this case, Luau will use the `return x > 0` statement to infer that `ispositive()` is a function taking a number and returning a boolean. Note that in this case, it was not necessary to add any explicit type annotations.

Based on Luau's type inference, the analysis tool will now flag the incorrect call to `ispositive()`:

```
$ luau-analyze test.luau
test.luau(7,18): TypeError: Type 'string' could not be converted into 'number'
```

Annotations

You can add annotations to locals, arguments, and function return types. Among other things, annotations can help enforce that you don't accidentally do something stupid. Here's how we would add annotations to `ispositive()`:

```
--!strict

function ispositive(x : number) : boolean
    return x > 0
end

local result : boolean
result = ispositive(1)
```

Now we've told explicitly told Luau that `ispositive()` accepts a number and returns a boolean. This wasn't strictly (pun intended) necessary in this case, because Luau's inference was able to deduce this already. But even in this case, there are advantages to explicit annotations. Imagine that later we decide to change `ispositive()` to return a string value:

```
--!strict

function ispositive(x : number) : boolean
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : boolean
result = ispositive(1)
```

Oops -- we're returning string values, but we forgot to update the function return type. Since we've told Luau that `ispositive()` returns a boolean (and that's how we're using it), the call site isn't flagged as an error. But because the annotation doesn't match our code, we get a warning in the function body itself:

```
$ luau-analyze test.luau
test.luau(5,9): TypeError: Type 'string' could not be converted into 'boolean'
test.luau(7,9): TypeError: Type 'string' could not be converted into 'boolean'
```

The fix is simple; just change the annotation to declare the return type as a string:

```
--!strict

function ispositive(x : number) : string
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : boolean
result = ispositive(1)
```

Well, almost - since we declared `result` as a boolean, the call site is now flagged:

```
$ luau-analyze test.luau
test.luau(12,10): TypeError: Type 'string' could not be converted into 'boolean'
```

If we update the type of the local variable, everything is good. Note that we could also just let Luau infer the type of `result` by changing it to the single line version `local result = ispositive(1)`.

```
--!strict

function ispositive(x : number) : string
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : string
result = ispositive(1)
```

Conclusions

This has been a brief tour of the basic functionality of Luau, but there's lots more to explore. If you're interested in reading more, check out our main reference pages for syntax (syntax) and typechecking (typecheck).

---

permalink: /grammar title: Grammar classes: wide

This is the complete syntax grammar for Luau in EBNF. More information about the terminal nodes String and Number is available in the syntax section (syntax).

```
chunk = block
block = {stat [';']} [laststat [';']]
stat = varlist '=' explist |
    var compoundop exp |
    functioncall |
    'do' block 'end' |
    'while' exp 'do' block 'end' |
    'repeat' block 'until' exp |
    'if' exp 'then' block {'elseif' exp 'then' block} ['else' block] 'end' |
    'for' binding '=' exp ',' exp [',' exp] 'do' block 'end' |
    'for' bindinglist 'in' explist 'do' block 'end' |
    'function' funcname funcbody |
    'local' 'function' NAME funcbody |
    'local' bindinglist ['=' explist] |
    ['export'] 'type' NAME ['<' GenericTypeListWithDefaults '>'] '=' Type

laststat = 'return' [explist] | 'break' | 'continue'

funcname = NAME {'.' NAME} [':' NAME]
funcbody = ['<' GenericTypeList '>'] '(' [parlist] ')' [':' ReturnType] block 'end'
parlist = bindinglist [',' '...'] | '...' [':' (Type | GenericTypePack)]

explist = {exp ','} exp
namelist = NAME {',' NAME}

binding = NAME [':' Type]
bindinglist = binding [',' bindinglist] (* equivalent of Lua 5.1 'namelist', except with optional type annotations *)

var = NAME | prefixexp '[' exp ']' | prefixexp '.' NAME
varlist = var {',' var}
prefixexp = var | functioncall | '(' exp ')'
functioncall = prefixexp funcargs | prefixexp ':' NAME funcargs

exp = asexp { binop exp } | unop exp { binop exp }
ifelseexp = 'if' exp 'then' exp {'elseif' exp 'then' exp} 'else' exp
asexp = simpleexp ['::' Type]
stringinterp = INTERP_BEGIN exp { INTERP_MID exp } INTERP_END
simpleexp = NUMBER | STRING | 'nil' | 'true' | 'false' | '...' | tableconstructor | 'function' funcbody | prefixexp | ifelseexp | st
funcargs =  '(' [explist] ')' | tableconstructor | STRING

tableconstructor = '{' [fieldlist] '}'
fieldlist = field {fieldsep field} [fieldsep]
field = '[' exp ']' '=' exp | NAME '=' exp | exp
fieldsep = ',' | ';'

compoundop :: '+=' | '-=' | '*=' | '/=' | '%=' | '^=' | '..='
binop = '+' | '-' | '*' | '/' | '^' | '%' | '..' | '<' | '<=' | '>' | '>=' | '==' | '~=' | 'and' | 'or'
unop = '-' | 'not' | '#'

SimpleType =
    'nil' |
    SingletonType |
    NAME ['.' NAME] [ '<' [TypeParams] '>' ] |
    'typeof' '(' exp ')' |
    TableType |
    FunctionType |
    '(' Type ')'

SingletonType = STRING | 'true' | 'false'

UnionSuffix = {'?'} {'|' SimpleType {'?'}}
IntersectionSuffix = {'&' SimpleType}
Type = SimpleType (UnionSuffix | IntersectionSuffix)

GenericTypePackParameter = NAME '...'
```

```
GenericTypeList = NAME [',' GenericTypeList] | GenericTypePackParameter {',' GenericTypePackParameter}

GenericTypePackParameterWithDefault = NAME '...' '=' (TypePack | VariadicTypePack | GenericTypePack)
GenericTypeListWithDefaults =
    GenericTypeList {',' GenericTypePackParameterWithDefault} |
    NAME {',' NAME} {',' NAME '=' Type} {',' GenericTypePackParameterWithDefault} |
    NAME '=' Type {',' GenericTypePackParameterWithDefault} |
    GenericTypePackParameterWithDefault {',' GenericTypePackParameterWithDefault}

TypeList = Type [',' TypeList] | '...' Type
BoundTypeList = [NAME ':'] Type [',' BoundTypeList] | '...' Type
TypeParams = (Type | TypePack | VariadicTypePack | GenericTypePack) [',' TypeParams]
TypePack = '(' [TypeList] ')'
GenericTypePack = NAME '...'
VariadicTypePack = '...' Type
ReturnType = Type | TypePack
TableIndexer = '[' Type ']' ':' Type
TableProp = NAME ':' Type
TablePropOrIndexer = TableProp | TableIndexer
PropList = TablePropOrIndexer {fieldsep TablePropOrIndexer} [fieldsep]
TableType = '{' Type '}' | '{' [PropList] '}'
FunctionType = ['<' GenericTypeList '>'] '(' [BoundTypeList] ')' '->' ReturnType
```

---

permalink: /library title: Library toc: true

Luau comes equipped with a standard library of functions designed to manipulate the built-in data types. Note that the library is relatively minimal and doesn't expose ways for scripts to interact with the host environment - it's expected that embedding applications provide extra functionality on top of this and limit or sandbox the system access appropriately, if necessary. For example, Roblox provides a rich API to interact with the 3D environment and limited APIs to interact with external services (https://developer.roblox.com/en-us/api-reference).

This page documents the available builtin libraries and functions. All of these are accessible by default by any script, assuming the host environment exposes them (which is usually a safe assumption outside of extremely constrained environments).

Global functions

While most library functions are provided as part of a library like `table`, a few global functions are exposed without extra namespacing.

```
function assert<T>(value: T, message: string?): T
```

`assert` checks if the value is truthy; if it's not (which means it's `false` or `nil`), it raises an error. The error message can be customized with an optional parameter. Upon success the function returns the `value` argument.

```
function error(obj: any, level: number?)
```

`error` raises an error with the specified object. Note that errors don't have to be strings, although they often are by convention; various error handling mechanisms like `pcall` preserve the error type. When `level` is specified, the error raised is turned into a string that contains call frame information for the caller at level `level`, where `1` refers to the function that called `error`. This can be useful to attribute the errors to callers, for example `error("Expected a valid object", 2)` highlights the caller of the function that called `error` instead of the function itself in the callstack.

```
function gcinfo(): number
```

`gcinfo` returns the total heap size in kilobytes, which includes bytecode objects, global tables as well as the script-allocated objects. Note that Luau uses an incremental garbage collector, and as such at any given point in time the heap may contain both reachable and unreachable objects. The number returned by `gcinfo` reflects the current heap consumption from the operating system perspective and can fluctuate over time as garbage collector frees objects.

```
function getfenv(target: (function | number)?): table
```

Returns the environment table for target function; when `target` is not a function, it must be a number corresponding to the caller stack index, where 1 means the function that calls `getfenv`, and the environment table is returned for the corresponding function from the call stack. When `target` is omitted it defaults to `1`, so `getfenv()` returns the environment table for the calling function.

```
function getmetatable(obj: any): table?
```

Returns the metatable for the specified object; when object is not a table or a userdata, the returned metatable is shared between all objects of the same type. Note that when metatable is protected (has a `__metatable` key), the value corresponding to that key is returned instead and may not be a table.

```
function next<K, V>(t: { [K]: V }, i: K?): (K, V)?
```

Given the table `t`, returns the next key-value pair after `i` in the table traversal order, or nothing if `i` is the last key. When `i` is `nil`, returns the first key-value pair instead.

```
function newproxy(mt: boolean?): userdata
```

Creates a new untyped userdata object; when `mt` is true, the new object has an empty metatable that can be modified using `getmetatable`.

```
function print(args: ...any)
```

Prints all arguments to the standard output, using Tab as a separator.

```
function rawequal(a: any, b: any): boolean
```

Returns true iff `a` and `b` have the same type and point to the same object (for garbage collected types) or are equal (for value types).

```
function rawget<K, V>(t: { [K]: V }, k: K): V?
```

Performs a table lookup with index `k` and returns the resulting value, if present in the table, or nil. This operation bypasses metatables/`__index`.

```
function rawset<K, V>(t: { [K] : V }, k: K, v: V)
```

Assigns table field `k` to the value `v`. This operation bypasses metatables/`__newindex`.

```
function select<T>(i: string, args: ...T): number
function select<T>(i: number, args: ...T): ...T
```

When called with `'#'` as the first argument, returns the number of remaining parameters passed. Otherwise, returns the subset of parameters starting with the specified index. Index can be specified from the start of the arguments (using 1 as the first argument), or from the end (using -1 as the last argument).

```
function setfenv(target: function | number, env: table)
```

Changes the environment table for target function to `env`; when `target` is not a function, it must be a number corresponding to the caller stack index, where 1 means the function that calls `setfenv`, and the environment table is returned for the corresponding function from the call stack.

```
function setmetatable(t: table, mt: table?)
```

Changes metatable for the given table. Note that unlike `getmetatable`, this function only works on tables. If the table already has a protected metatable (has a `__metatable` field), this function errors.

```
function tonumber(s: string, base: number?): number?
```

Converts the input string to the number in base `base` (default 10) and returns the resulting number. If the conversion fails (that is, if the input string doesn't represent a valid number in the specified base), returns `nil` instead.

```
function tostring(obj: any): string
```

Converts the input object to string and returns the result. If the object has a metatable with `__tostring` field, that method is called to perform the conversion.

```
function type(obj: any): string
```

Returns the type of the object, which is one of `"nil"`, `"boolean"`, `"number"`, `"vector"`, `"string"`, `"table"`, `"function"`, `"userdata"` or `"thread"`.

```
function typeof(obj: any): string
```

Returns the type of the object; for userdata objects that have a metatable with the `__type` field *and* are defined by the host (not `newproxy`), returns the value for that key. For custom userdata objects, such as ones returned by `newproxy`, this function returns `"userdata"` to make sure host-defined types can not be spoofed.

```
function ipairs(t: table): <iterator>
```

Returns the triple (generator, state, nil) that can be used to traverse the table using a `for` loop. The traversal results in key-value pairs for the numeric portion of the table; key starts from 1 and increases by 1 on each iteration. The traversal terminates when reaching the first `nil` value (so `ipairs` can't be used to traverse array-like tables with holes).

```
function pairs(t: table): <iterator>
```

Returns the triple (generator, state, nil) that can be used to traverse the table using a `for` loop. The traversal results in key-value pairs for all keys in the table, numeric and otherwise, but doesn't have a defined order.

```
function pcall(f: function, args: ...any): (boolean, ...any)
```

Calls function `f` with parameters `args`. If the function succeeds, returns `true` followed by all return values of `f`. If the function raises an error, returns `false` followed by the error object. Note that `f` can yield, which results in the entire coroutine yielding as well.

```
function xpcall(f: function, e: function, args: ...any): (boolean, ...any)
```

Calls function `f` with parameters `args`. If the function succeeds, returns `true` followed by all return values of `f`. If the function raises an error, calls `e` with the error object as an argument, and returns `false` followed by all return values of `e`. Note that `f` can yield, which results in the entire coroutine yielding as well. `e` can neither yield nor error - if it does raise an error, `xpcall` returns with `false` followed by a special error message.

```
function unpack<V>(a: {V}, f: number?, t: number?): ...V
```

Returns all values of `a` with indices in `[f..t]` range. `f` defaults to 1 and `t` defaults to `#a`. Note that this is equivalent to `table.unpack`.

math library

```
function math.abs(n: number): number
```

Returns the absolute value of `n`. Returns NaN if the input is NaN.

```
function math.acos(n: number): number
```

Returns the arc cosine of `n`, expressed in radians. Returns a value in `[0, pi]` range. Returns NaN if the input is not in `[-1, +1]` range.

```
function math.asin(n: number): number
```

Returns the arc sine of `n`, expressed in radians. Returns a value in `[-pi/2, +pi/2]` range. Returns NaN if the input is not in `[-1, +1]` range.

```
function math.atan2(y: number, x: number): number
```

Returns the arc tangent of `y/x`, expressed in radians. The function takes into account the sign of both arguments in order to determine the quadrant. Returns a value in `[-pi, pi]` range.

```
function math.atan(n: number): number
```

Returns the arc tangent of `n`, expressed in radians. Returns a value in `[-pi/2, pi-2]` range.

```
function math.ceil(n: number): number
```

Rounds `n` upwards to the next integer boundary.

```
function math.cosh(n: number): number
```

Returns the hyperbolic cosine of `n`.

```
function math.cos(n: number): number
```

Returns the cosine of `n`, which is an angle in radians. Returns a value in `[0, 1]` range.

```
function math.deg(n: number): number
```

Converts `n` from radians to degrees and returns the result.

```
function math.exp(n: number): number
```

Returns the base-e exponent of `n`, that is `e^n`.

```
function math.floor(n: number): number
```

Rounds `n` downwards to previous integer boundary.

```
function math.fmod(x: number, y: number): number
```

Returns the remainder of `x` modulo `y`, rounded towards zero. Returns NaN if `y` is zero.

```
function math.frexp(n: number): (number, number)
```

Splits the number into a significand (a number in `[-1, +1]` range) and binary exponent such that `n = s * 2^e`, and returns `s, e`.

```
function math.ldexp(s: number, e: number): number
```

Given the significand and a binary exponent, returns a number `s * 2^e`.

```
function math.log10(n: number): number
```

Returns base-10 logarithm of the input number. Returns NaN if the input is negative, and negative infinity if the input is 0. Equivalent to `math.log(n, 10)`.

```
function math.log(n: number, base: number?): number
```

Returns logarithm of the input number in the specified base; base defaults to `e`. Returns NaN if the input is negative, and negative infinity if the input is 0.

```
function math.max(list: ...number): number
```

Returns the maximum number of the input arguments. The function requires at least one input and will error if zero parameters are passed. If one of the inputs is a NaN, the result may or may not be a NaN.

```
function math.min(list: ...number): number
```

Returns the minimum number of the input arguments. The function requires at least one input and will error if zero parameters are passed. If one of the inputs is a NaN, the result may or may not be a NaN.

```
function math.modf(n: number): (number, number)
```

Returns the integer and fractional part of the input number. Both the integer and fractional part have the same sign as the input number, e.g. `math.modf(-1.5)` returns `-1, -0.5`.

```
function math.pow(x: number, y: number): number
```

Returns `x` raised to the power of `y`.

```
function math.rad(n: number): number
```

Converts `n` from degrees to radians and returns the result.

```
function math.random(): number
function math.random(n: number): number
function math.random(min: number, max: number): number
```

Returns a random number using the global random number generator. A zero-argument version returns a number in `[0, 1]` range. A one-argument version returns a number in `[1, n]` range. A two-argument version returns a number in `[min, max]` range. The input arguments are truncated to integers, so `math.random(1.5)` always returns 1.

```
function math.randomseed(seed: number)
```

Reseeds the global random number generator; subsequent calls to `math.random` will generate a deterministic sequence of numbers that only depends on `seed`.

```
function math.sinh(n: number): number
```

Returns a hyperbolic sine of `n`.

```
function math.sin(n: number): number
```

Returns the sine of `n`, which is an angle in radians. Returns a value in `[0, 1]` range.

```
function math.sqrt(n: number): number
```

Returns the square root of `n`. Returns NaN if the input is negative.

```
function math.tanh(n: number): number
```

Returns the hyperbolic tangent of `n`.

```
function math.tan(n: number): number
```

Returns the tangent of `n`, which is an angle in radians.

```
function math.noise(x: number, y: number?, z: number?): number
```

Returns 3D Perlin noise value for the point `(x, y, z)` (`y` and `z` default to zero if absent). Returns a value in `[-1, 1]` range.

```
function math.clamp(n: number, min: number, max: number): number
```

Returns `n` if the number is in `[min, max]` range; otherwise, returns `min` when `n < min`, and `max` otherwise. If `n` is NaN, may or may not return NaN. The function errors if `min > max`.

```
function math.sign(n: number): number
```

Returns `-1` if `n` is negative, `1` if `n` is positive, and `0` if `n` is zero or NaN.

```
function math.round(n: number): number
```

Rounds `n` to the nearest integer boundary. If `n` is exactly halfway between two integers, rounds `n` away from 0.

table library

```
function table.concat(a: {string}, sep: string?, f: number?, t: number?): string
```

Concatenate all elements of `a` with indices in range `[f..t]` together, using `sep` as a separator if present. `f` defaults to 1 and `t` defaults to `#a`.

```
function table.foreach<K, V, R>(t: { [K]: V }, f: (K, V) -> R?): R?
```

Iterates over all elements of the table in unspecified order; for each key-value pair, calls `f` and returns the result of `f` if it's non-nil. If all invocations of `f` returned `nil`, returns no values. This function has been deprecated and is not recommended for use in new code; use `for` loop instead.

```
function table.foreachi<V, R>(t: {V}, f: (number, V) -> R?): R?
```

Iterates over numeric keys of the table in `[1..#t]` range in order; for each key-value pair, calls `f` and returns the result of `f` if it's non-nil. If all invocations of `f` returned `nil`, returns no values. This function has been deprecated and is not recommended for use in new code; use `for` loop instead.

```
function table.getn<V>(t: {V}): number
```

Returns the length of table `t`. This function has been deprecated and is not recommended for use in new code; use `#t` instead.

```
function table.maxn<V>(t: {V}): number
```

Returns the maximum numeric key of table `t`, or zero if the table doesn't have numeric keys.

```
function table.insert<V>(t: {V}, v: V)
function table.insert<V>(t: {V}, i: number, v: V)
```

When using a two-argument version, appends the value to the array portion of the table (equivalent to `t[#t+1] = v`). When using a three-argument version, inserts the value at index `i` and shifts values at indices after that by 1. `i` should be in `[1..#t]` range.

```
function table.remove<V>(t: {V}, i: number?): V?
```

Removes element `i` from the table and shifts values at indices after that by 1. If `i` is not specified, removes the last element of the table. `i` should be in `[1..#t]` range. Returns the value of the removed element, or `nil` if no element was removed (e.g. table was empty).

```
function table.sort<V>(t: {V}, f: ((V, V) -> boolean)?)
```

Sorts the table `t` in ascending order, using `f` as a comparison predicate: `f` should return `true` iff the first parameter should be before the second parameter in the resulting table. When `f` is not specified, builtin less-than comparison is used instead. The comparison predicate must establish a strict weak ordering - sort results are undefined otherwise.

```
function table.pack<V>(args: ...V): { [number]: V, n: number }
```

Returns a table that consists of all input arguments as array elements, and `n` field that is set to the number of inputs.

```
function table.unpack<V>(a: {V}, f: number?, t: number?): ...V
```

Returns all values of `a` with indices in `[f..t]` range. `f` defaults to 1 and `t` defaults to `#a`.

```
function table.move<V>(a: {V}, f: number, t: number, d: number, tt: {V}?)
```

Copies elements in range `[f..t]` from table `a` to table `tt` if specified and `a` otherwise, starting from the index `d`.

```
function table.create<V>(n: number, v: V?): {V}
```

Creates a table with `n` elements; all of them (range `[1..n]`) are set to `v`. When `v` is nil or omitted, the returned table is empty but has preallocated space for `n` elements which can make subsequent insertions faster. Note that preallocation is only performed for the array portion of the table - using `table.create` on dictionaries is counter-productive.

```
function table.find<V>(t: {V}, v: V): number?
```

Find the first element in the table that is equal to `v` and returns its index; the traversal stops at the first `nil`. If the element is not found, `nil` is returned instead.

```
function table.clear(t: table)
```

Removes all elements from the table while preserving the table capacity, so future assignments don't need to reallocate space.

```
function table.freeze(t: table): table
```

Given a non-frozen table, freezes it such that all subsequent attempts to modify the table or assign its metatable raise an error. If the input table is already frozen or has a protected metatable, the function raises an error; otherwise it returns the input table. Note that the table is frozen in-place and is not being copied. Additionally, only `t` is frozen, and keys/values/metatable of `t` don't change their state and need to be frozen separately if desired.

```
function table.isfrozen(t: table): boolean
```

Returns `true` iff the input table is frozen.

```
function table.clone(t: table): table
```

Returns a copy of the input table that has the same metatable, same keys and values, and is not frozen even if `t` was. The copy is shallow: implementing a deep recursive copy automatically is challenging, and often only certain keys need to be cloned recursively which can be done after the initial clone by modifying the resulting table.

string library

```
function string.byte(s: string, f: number?, t: number?): ...number
```

Returns the numeric code of every byte in the input string with indices in range `[f..t]`. `f` defaults to 1 and `t` defaults to `f`, so a two-argument version of this function returns a single number. If the function is called with a single argument and the argument is out of range, the function returns no values.

```
function string.char(args: ...number): string
```

Returns the string that contains a byte for every input number; all inputs must be integers in `[0..255]` range.

```
function string.find(s: string, p: string, init: number?, plain: boolean?): (number?, number?, ...string)
```

Tries to find an instance of pattern `p` in the string `s`, starting from position `init` (defaults to 1). When `plain` is true, the search is using raw (case-sensitive) string equality, otherwise `p` should be a string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1). If a match is found, returns the position of the match and the length of the match, followed by the pattern captures; otherwise returns `nil`.

```
function string.format(s: string, args: ...any): string
```

Returns a formatted version of the input arguments using a printf-style format string (https://en.cppreference.com/w/c/io/fprintf) `s`. The following format characters are supported:

- `c`: expects an integer number and produces a character with the corresponding character code
- `d`, `i`, `u`: expects an integer number and produces the decimal representation of that number
- `o`: expects an integer number and produces the octal representation of that number
- `x`, `X`: expects an integer number and produces the hexadecimal representation of that number, using lower case or upper case hexadecimal characters
- `e`, `E`, `f`, `g`, `G`: expects a number and produces the floating point representation of that number, using scientific or decimal representation
- `q`: expects a string and produces the same string quoted using double quotation marks, with escaped special characters if necessary
- `s`: expects a string and produces the same string verbatim

The formats support modifiers `-`, `+`, space, `#` and `0`, as well as field width and precision modifiers - with the exception of `*`.

```
function string.gmatch(s: string, p: string): <iterator>
```

Produces an iterator function that, when called repeatedly explicitly or via `for` loop, produces matches of string `s` with string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1) `p`. For every match, the captures within the pattern are returned if present (if a pattern has no captures, the entire matching substring is returned instead).

```
function string.gsub(s: string, p: string, f: function | table | string, maxs: number?): (string, number)
```

For every match of string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1) `p` in `s`, replace the match according to `f`. The substitutions stop after the limit of `maxs`, and the function returns the resulting string followed by the number of substitutions.

When `f` is a string, the substitution uses the string as a replacement. When `f` is a table, the substitution uses the table element with key corresponding to the first pattern capture, if present, and entire match otherwise. Finally, when `f` is a function, the substitution uses the result of calling `f` with call pattern captures, or entire matching substring if no captures are present.

```
function string.len(s: string): number
```

Returns the number of bytes in the string (equivalent to `#s`).

```
function string.lower(s: string): string
```

Returns a string where each byte corresponds to the lower-case ASCII version of the input byte in the source string.

```
function string.match(s: string, p: string, init: number?): ...string?
```

Tries to find an instance of pattern `p` in the string `s`, starting from position `init` (defaults to 1). `p` should be a string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1). If a match is found, returns all pattern captures, or entire matching substring if no captures are present, otherwise returns `nil`.

```
function string.rep(s: string, n: number): string
```

Returns the input string `s` repeated `n` times. Returns an empty string if `n` is zero or negative.

```
function string.reverse(s: string): string
```

Returns the string with the order of bytes reversed compared to the original. Note that this only works if the input is a binary or ASCII string.

```
function string.sub(s: string, f: number, t: number?): string
```

Returns a substring of the input string with the byte range `[f..t]`; `t` defaults to `#s`, so a two-argument version returns a string suffix.

```
function string.upper(s: string): string
```

Returns a string where each byte corresponds to the upper-case ASCII version of the input byte in the source string.

```
function string.split(s: string, sep: string?): {string}
```

Splits the input string using `sep` as a separator (defaults to `","`) and returns the resulting substrings. If separator is empty, the input string is split into separate one-byte strings.

```
function string.pack(f: string, args: ...any): string
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), encodes all input parameters according to the packing format and returns the resulting string. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

```
function string.packsize(f: string): number
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), returns the size of the resulting packed representation. The pack format can't use variable-length format specifiers. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

```
function string.unpack(f: string, s: string): ...any
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), decodes the input string according to the packing format and returns all resulting values. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

coroutine library

```
function coroutine.create(f: function): thread
```

Returns a new coroutine that, when resumed, will run function `f`.

```
function coroutine.running(): thread?
```

Returns the currently running coroutine, or `nil` if the code is running in the main coroutine (depending on the host environment setup, main coroutine may never be used for running code).

```
function coroutine.status(co: thread): string
```

Returns the status of the coroutine, which can be `"running"`, `"suspended"`, `"normal"` or `"dead"`. Dead coroutines have finished their execution and can not be resumed, but their state can still be inspected as they are not dead from the garbage collector point of view.

```
function coroutine.wrap(f: function): function
```

Creates a new coroutine and returns a function that, when called, resumes the coroutine and passes all arguments along to the suspension point. When the coroutine yields or finishes, the wrapped function returns with all values returned at the suspension point.

```
function coroutine.yield(args: ...any): ...any
```

Yields the currently running coroutine and passes all arguments along to the code that resumed the coroutine. The coroutine becomes suspended; when the coroutine is resumed again, the resumption arguments will be forwarded to `yield` which will behave as if it returned all of them.

```
function coroutine.isyieldable(): boolean
```

Returns `true` iff the currently running coroutine can yield. Yielding is prohibited when running inside metamethods like `__index` or C functions like `table.foreach` callback, with the exception of `pcall`/`xpcall`.

```
function coroutine.resume(co: thread, args: ...any): (boolean, ...any)
```

Resumes the coroutine and passes the arguments along to the suspension point. When the coroutine yields or finishes, returns `true` and all values returned at the suspension point. If an error is raised during coroutine resumption, this function returns `false` and the error object, similarly to `pcall`.

```
function coroutine.close(co: thread): (boolean, any?)
```

Closes the coroutine which puts coroutine in the dead state. The coroutine must be dead or suspended - in particular it can't be currently running. If the coroutine that's being closed was in an error state, returns `false` along with an error object; otherwise returns `true`. After closing, the coroutine can't be resumed and the coroutine stack becomes empty.

bit32 library

All functions in the `bit32` library treat input numbers as 32-bit unsigned integers in `[0..4294967295]` range. The bit positions start at 0 where 0 corresponds to the least significant bit.

```
function bit32.arshift(n: number, i: number): number
```

Shifts `n` by `i` bits to the right (if `i` is negative, a left shift is performed instead). The most significant bit of `n` is propagated during the shift. When `i` is larger than 31, returns an integer with all bits set to the sign bit of `n`. When `i` is smaller than `-31`, 0 is returned.

```
function bit32.band(args: ...number): number
```

Performs a bitwise `and` of all input numbers and returns the result. If the function is called with no arguments, an integer with all bits set to 1 is returned.

```
function bit32.bnot(n: number): number
```

Returns a bitwise negation of the input number.

```
function bit32.bor(args: ...number): number
```

Performs a bitwise `or` of all input numbers and returns the result. If the function is called with no arguments, zero is returned.

```
function bit32.bxor(args: ...number): number
```

Performs a bitwise `xor` (exclusive or) of all input numbers and returns the result. If the function is called with no arguments, zero is returned.

```
function bit32.btest(args: ...number): boolean
```

Perform a bitwise `and` of all input numbers, and return `true` iff the result is not 0. If the function is called with no arguments, `true` is returned.

```
function bit32.extract(n: number, f: number, w: number?): number
```

Extracts bits of `n` at position `f` with a width of `w`, and returns the resulting integer. `w` defaults to `1`, so a two-argument version of `extract` returns the bit value at position `f`. Bits are indexed starting at 0. Errors if `f` and `f+w-1` are not between 0 and 31.

```
function bit32.lrotate(n: number, i: number): number
```

Rotates n to the left by i bits (if i is negative, a right rotate is performed instead); the bits that are shifted past the bit width are shifted back from the right.

```
function bit32.lshift(n: number, i: number): number
```

Shifts n to the left by i bits (if i is negative, a right shift is performed instead). When i is outside of `[-31..31]` range, returns 0.

```
function bit32.replace(n: number, r: number, f: number, w: number?): number
```

Replaces bits of n at position f and width w with r, and returns the resulting integer. w defaults to 1, so a three-argument version of `replace` changes one bit at position f to r (which should be 0 or 1) and returns the result. Bits are indexed starting at 0. Errors if f and `f+w-1` are not between 0 and 31.

```
function bit32.rrotate(n: number, i: number): number
```

Rotates n to the right by i bits (if i is negative, a left rotate is performed instead); the bits that are shifted past the bit width are shifted back from the left.

```
function bit32.rshift(n: number, i: number): number
```

Shifts n to the right by i bits (if i is negative, a left shift is performed instead). When i is outside of `[-31..31]` range, returns 0.

```
function bit32.countlz(n: number): number
```

Returns the number of consecutive zero bits in the 32-bit representation of n starting from the left-most (most significant) bit. Returns 32 if n is zero.

```
function bit32.countrz(n: number): number
```

Returns the number of consecutive zero bits in the 32-bit representation of n starting from the right-most (least significant) bit. Returns 32 if n is zero.

```
function bit32.byteswap(n: number): number
```

Returns n with the order of the bytes swappped.

utf8 library

Strings in Luau can contain arbitrary bytes; however, in many applications strings representing text contain UTF8 encoded data by convention, that can be inspected and manipulated using `utf8` library.

```
function utf8.offset(s: string, n: number, i: number?): number?
```

Returns the byte offset of the Unicode codepoint number n in the string, starting from the byte position i. When the character is not found, returns `nil` instead.

```
function utf8.codepoint(s: string, i: number?, j: number?): ...number
```

Returns a number for each Unicode codepoint in the string with the starting byte offset in `[i..j]` range. i defaults to 1 and j defaults to i, so a two-argument version of this function returns the Unicode codepoint that starts at byte offset i.

```
function utf8.char(args: ...number): string
```

Creates a string by concatenating Unicode codepoints for each input number.

```
function utf8.len(s: string, i: number?, j: number?): number?
```

Returns the number of Unicode codepoints with the starting byte offset in `[i..j]` range, or `nil` followed by the first invalid byte position if the input string is malformed. i defaults to 1 and j defaults to `#s`, so `utf8.len(s)` returns the number of Unicode codepoints in string s or `nil` if the string is malformed.

```
function utf8.codes(s: string): <iterator>
```

Returns an iterator that, when used in `for` loop, produces the byte offset and the codepoint for each Unicode codepoints that s consists of.

os library

```
function os.clock(): number
```

Returns a high-precision timestamp (in seconds) that doesn't have a defined baseline, but can be used to measure duration with sub-microsecond precision.

```
function os.date(s: string?, t: number?): table | string
```

Returns the table or string representation of the time specified as `t` (defaults to current time) according to `s` format string.

When `s` starts with `!`, the result uses UTC, otherwise it uses the current timezone.

If `s` is equal to `*t` (or `!*t`), a table representation of the date is returned, with keys `sec`/`min`/`hour` for the time (using 24-hour clock), `day`/`month`/`year` for the date, `wday` for week day (1..7), `yday` for year day (1..366) and `isdst` indicating whether the timezone is currently using daylight savings.

Otherwise, `s` is interpreted as a date format string (https://www.cplusplus.com/reference/ctime/strftime/), with the valid specifiers including any of `aAbBcdHIjmMpSUwWxXyYzZ` or `%`. `s` defaults to `"%c"` so `os.date()` returns the human-readable representation of the current date in local timezone.

```
function os.difftime(a: number, b: number): number
```

Calculates the difference in seconds between `a` and `b`; provided for compatibility only. Please use `a - b` instead.

```
function os.time(t: table?): number
```

When called without arguments, returns the current date/time as a Unix timestamp. When called with an argument, expects it to be a table that contains `sec`/`min`/`hour`/`day`/`month`/`year` keys and returns the Unix timestamp of the specified date/time in UTC.

debug library

```
function debug.info(co: thread, level: number, s: string): ...any
function debug.info(level: number, s: string): ...any
function debug.info(f: function, s: string): ...any
```

Given a stack frame or a function, and a string that specifies the requested information, returns the information about the stack frame or function.

Each character of `s` results in additional values being returned in the same order as the characters appear in the string:

- `s` returns source path for the function
- `l` returns the line number for the stack frame or the line where the function is defined when inspecting a function object
- `n` returns the name of the function, or an empty string if the name is not known
- `f` returns the function object itself
- `a` returns the number of arguments that the function expects followed by a boolean indicating whether the function is variadic or not

For example, `debug.info(2, "sln")` returns source file, current line and function name for the caller of the current function.

```
function debug.traceback(co: thread, msg: string?, level: number?): string
function debug.traceback(msg: string?, level: number?): string
```

Produces a stringified callstack of the given thread, or the current thread, starting with level `level`. If `msg` is specified, then the resulting callstack includes the string before the callstack output, separated with a newline. The format of the callstack is human-readable and subject to change.

buffer library

Buffer is an object that represents a fixed-size mutable block of memory.

All operations on a buffer are provided using the 'buffer' library functions.

Many of the functions accept an offset in bytes from the start of the buffer. Offset of 0 from the start of the buffer memory block accesses the first byte.

All offsets, counts and sizes should be non-negative integer numbers.

If the bytes that are accessed by any read or write operation are outside the buffer memory, an error is thrown.

```
function buffer.create(size: number): buffer
```

Creates a buffer of the requested size with all bytes initialized to 0.

Size limit is 1GB or 1,073,741,824 bytes.

```
function buffer.fromstring(str: string): buffer
```

Creates a buffer initialized to the contents of the string.

The size of the buffer equals to the length of the string.

```
function buffer.tostring(b: buffer): string
```

Returns the buffer data as a string.

```
function buffer.len(b: buffer): number
```

Returns the size of the buffer in bytes.

```
function buffer.readi8(b: buffer, offset: number): number
function buffer.readu8(b: buffer, offset: number): number
function buffer.readi16(b: buffer, offset: number): number
function buffer.readu16(b: buffer, offset: number): number
function buffer.readi32(b: buffer, offset: number): number
function buffer.readu32(b: buffer, offset: number): number
function buffer.readf32(b: buffer, offset: number): number
function buffer.readf64(b: buffer, offset: number): number
```

Used to read the data from the buffer by reinterpreting bytes at the offset as the type in the argument and converting it into a number.

Available types:

Function	Type	Range
readi8	signed 8-bit integer	[-128, 127]
readu8	unsigned 8-bit integer	[0, 255]
readi16	signed 16-bit integer	[-32,768, 32,767]
readu16	unsigned 16-bit integer	[0, 65,535]
readi32	signed 32-bit integer	[-2,147,483,648, 2,147,483,647]
readu32	unsigned 32-bit integer	[0, 4,294,967,295]
readf32	32-bit floating-point number	Single-precision IEEE 754 number
readf64	64-bit floating-point number	Double-precision IEEE 754 number

Floating-point numbers are read and written using a format specified by IEEE 754.

If a floating-point value matches any of bit patterns that represent a NaN (not a number), returned value might be converted to a different quiet NaN representation.

Read and write operations use the little endian byte order.

Integer numbers are read and written using two's complement representation.

```
function buffer.writei8(b: buffer, offset: number, value: number): ()
function buffer.writeu8(b: buffer, offset: number, value: number): ()
function buffer.writei16(b: buffer, offset: number, value: number): ()
function buffer.writeu16(b: buffer, offset: number, value: number): ()
function buffer.writei32(b: buffer, offset: number, value: number): ()
function buffer.writeu32(b: buffer, offset: number, value: number): ()
function buffer.writef32(b: buffer, offset: number, value: number): ()
function buffer.writef64(b: buffer, offset: number, value: number): ()
```

Used to write data to the buffer by converting the number into the type in the argument and reinterpreting it as individual bytes.

Ranges of acceptable values can be seen in the table above.

When writing integers, the number is converted using `bit32` library rules.

Values that are out-of-range will take less significant bits of the full number. For example, writing 43,981 (0xabcd) using writei8 function will take 0xcd and interpret it as an 8-bit signed number -51. It is still recommended to keep all numbers in range of the target type.

Results of converting special number values (inf/nan) to integers are platform-specific.

```
function buffer.readstring(b: buffer, offset: number, count: number): string
```

Used to read a string of length 'count' from the buffer at specified offset.

```
function buffer.writestring(b: buffer, offset: number, value: string, count: number?): ()
```

Used to write data from a string into the buffer at a specified offset.

If an optional 'count' is specified, only 'count' bytes are taken from the string.

Count cannot be larger than the string length.

```
function buffer.copy(target: buffer, targetOffset: number, source: buffer, sourceOffset: number?, count: number?): ()
```

Copy 'count' bytes from 'source' starting at offset 'sourceOffset' into the 'target' at 'targetOffset'.

It is possible for 'source' and 'target' to be the same. Copying an overlapping region inside the same buffer acts as if the source region is copied into a temporary buffer and then that buffer is copied over to the target.

If 'sourceOffset' is nil or is omitted, it defaults to 0.

If 'count' is 'nil' or is omitted, the whole 'source' data starting from 'sourceOffset' is copied.

```
function buffer.fill(b: buffer, offset: number, value: number, count: number?): ()
```

Sets the 'count' bytes in the buffer starting at the specified 'offset' to the 'value'.

If 'count' is 'nil' or is omitted, all bytes from the specified offset until the end of the buffer are set.

---

layout: default title: LIMITED USE LICENSE permalink: /limited-terms-of-use

ROBLOX, INC.

LIMITED USE LICENSE

Roblox Corporation (“Roblox”) is making certain software for your access and use on a pre-release, trial, beta, evaluation, or similar limited purpose basis pursuant to this Limited Use License (the “License”). By clicking agree, downloading, accessing, or using the software and materials (which may include associated data and data sets, models, and documentation) accompanying this License (the “Licensed Software”), you agree to be bound by the terms and conditions of this License, including acknowledging that your rights are conditioned on compliance with these terms and revocable at any time, that the Licensed Software is made available on an “as-is” basis without warranty or guarantee of any kind, and that Roblox expressly reserves all rights not expressly granted in this License. Access and use of the Licensed Software may require applying for and maintaining an active registration with Roblox, the use of which account and associated services (the “Service”) is governed by and subject to separate terms (the “Terms of Use”).

1. **Evaluation Use of the Licensed Software.** Subject to your complete and ongoing compliance with this License (including any applicable Terms of Use), Roblox grants you a limited, nonexclusive, non-transferable, non-sublicensable, revocable permission to access and use the Licensed Software during the License Term in accordance with any applicable documentation, solely for the purpose of internal, non-production evaluation and testing of the Licensed Software, and solely in connection with your otherwise permitted use of the Roblox products and services for which the Licensed Software was designed.

2. **Source Code License.** Solely in the event that all or a portion of the Licensed Software is provided in source code form (the “Licensed Source”) the permission granted in Section 1 also includes the right to access and modify the Licensed Source, subject to the same limitations and restrictions as provided in Section 1 and elsewhere in this License and the Terms of Use. The modified version or other derivatives of such Licensed Source (in source or compiled form) will be deemed Licensed Software hereunder.

3. **Usage Guidelines.** You must not use the Licensed Software to transmit unlawful or otherwise objectionable material, transmit viruses or other harmful computer code, interfere with the performance of the Service or the data contained therein, attempt to gain unauthorized access to the Service or networks related to the Service, or interfere with another’s use of the Service. You must not distribute the Licensed Software or derivatives thereof in whole or in part, or modify, copy, or make derivative works based on, or otherwise reverse engineer, disassemble, or decompile any portion of the Licensed Software not made available to you by Roblox in source form. You may not access or use the Licensed Software or the Service in connection with any alternative or competitive service, or to otherwise reproduce, emulate, or replace features of the Service. You also may not use the Licensed Software to create malicious or abusive content (as determined by Roblox) or any content that violates a Roblox guideline or policy or use the Licensed Software in any manner that infringes, misappropriates, or otherwise violates any intellectual property right or other right of any person, or that violates any applicable laws.

4. **Feedback.** From time to time you may, but are not required to, provide Roblox with feedback regarding the Licensed Software, including by reporting any errors you encounter, providing suggestions for modifications and enhancements, submitting bug fixes, contributions, or other associated code, or making other similar submissions (collectively, "Feedback"). You retain ownership of your rights in Feedback. Notwithstanding, Feedback is provided to Roblox on a non-confidential basis (notwithstanding any indication to the contrary in any accompanying communication) and by submitting Feedback you hereby irrevocably grant Roblox and its successors an unrestricted, unlimited, worldwide, nonexclusive, transferable, sublicensable, fully paid up, royalty free, sublicensable (through multiple tiers) license to reproduce, distribute, publicly perform or display, and prepare derivatives of the Feedback, and to make, have made, use, sell, offer for sale, and import any product or service, and to practice any method disclosed in the Feedback, in any form or manner and without limitation, attribution, or compensation.

5. **Reservation of Rights.** Roblox retains all its right, title, and interest, including all intellectual property rights, in and to the Licensed Software and derivatives thereof, and nothing herein will be deemed to grant or confer on you any ownership of any such rights, whether expressly, by implication, estoppel, or otherwise. You acknowledge that Roblox may track usage of the Licensed Software in connection with the Services, and Roblox retains unrestricted rights to all related data, and any analytical results, models, methods or learnings derived therefrom.

6. **Additional Restrictions.** Your testing of the Licensed Software and the results thereof and your Feedback, are all Roblox confidential information ("Confidential Information") and may not be used for any purpose other than your testing and evaluation of the Licensed Software, or shared with any other person or entity. You may not (i) disclose, permit disclosure of, publish, or disseminate Confidential Information to anyone (ii) perform any benchmarking test or similar comparative research, or (iii) disclose any results, opinions, or summaries of the Licensed Software or any Confidential Information.

7. **DISCLAIMER; LIMITATION OF LIABILITY.** THE LICENSED SOFTWARE IS PROVIDED "AS IS" AND WITH ALL FAULTS. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, ROBLOX HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, NONINFRINGEMENT AND FITNESS FOR A PARTICULAR PURPOSE, OR THAT THE LICENSED SOFTWARE SHALL BE ERROR-FREE OR UNINTERRUPTED. IN NO EVENT SHALL ROBLOX HAVE ANY DIRECT OR INDIRECT LIABILITY TO YOU FOR ANY REASON UNDER THIS LICENSE, INCLUDING ANY LOST PROFITS, LOSS OF DATA, LOSS OF USE, OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY SPECIAL, PUNITIVE, OR EXEMPLARY DAMAGES HOWEVER CAUSED AND, WHETHER IN CONTRACT, TORT OR UNDER ANY OTHER THEORY OF LIABILITY, WHETHER OR NOT EITHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In addition, you agree and understand that the Licensed Software may be provided on an alpha, beta, or other pre-release basis, and Roblox is under no obligation to make any further releases, including any production release, and that any subsequent release may differ from the Licensed Software in any manner Roblox sees fit, including addition, modification, or removal of features and functionality present in the Licensed Software. You acknowledge that the limitations of liability and the disclaimers of warranties and damages set forth herein form an essential basis of the bargain between you and Roblox, and that they will survive and apply even if found to have failed of their essential purpose, to the greatest extent permissible pursuant to applicable law.

8. **Term and Termination.** This Agreement commences on the date you accept this License and, unless terminated earlier as described below or extended by mutual written agreement, shall terminate upon the later of the license term specified at the time you access the Licensed Software, or the public release of the production version of the Licensed Software (the "License Term"). Roblox may terminate the License (and your access to the Licensed Software and any associated materials) at any time for any reason or no reason. Any provisions that by their nature or express terms survive expiration or termination of this License will survive any expiration or termination of this License. Your obligations with respect to Confidential Information and Feedback will survive indefinitely, notwithstanding any eventual publication of a commercial or production version of the Licensed Software. Upon expiration or termination of this License for any reason, you must cease using the Licensed Software, and return or destroy copies of any Licensed Software or Licensed Source (and derivatives thereof) or other Confidential Information in your possession.

---

permalink: /lint title: Linting toc: true

Luau comes with a set of linting passes, that help make sure that the code is correct and consistent. Unlike the type checker, that models the behavior of the code thoroughly and points toward type mismatches that are likely to result in runtime errors, the linter is more opinionated and produces warnings that can often be safely ignored, although it's recommended to keep the code clean of the warnings.

Linter produces many different types of warnings; many of these are enabled by default, and can be suppressed by declaring `--!nolint NAME` at the top of the file. In dire situations `--!nolint` at the top of the file can be used to completely disable all warnings (note that the type checker is still active, and requires a separate `--!nocheck` declaration).

The rest of this page documents all warnings produced by the linter; each warning has a name and a numeric code, the latter is used when displaying warnings.

UnknownGlobal (1)

By default, variables in Luau are global (this is inherited from Lua 5.x and can't be changed because of backwards compatibility). This means that typos in identifiers are invisible to the parser, and often break at runtime. For this reason, the linter considers all globals that aren't part of the builtin global table and aren't explicitly defined in the script "unknown":

```
local displayName = "Roblox"

-- Unknown global 'displaName'
print(displaName)
```

Note that injecting globals via `setfenv` can produce this warning in correct code; global injection is incompatible with type checking and has performance implications so we recommend against it and in favor of using `require` with correctly scoped identifiers.

DeprecatedGlobal (2)

Some global names exist for compatibility but their use is discouraged. This mostly affects globals introduced by Roblox, and since they can have problematic behavior or can break in the future, this warning highlights their uses:

```
-- Global 'ypcall' is deprecated, use 'pcall' instead
ypcall(function()
    print("hello")
end)
```

GlobalUsedAsLocal (3)

The UnknownGlobal lint can catch typos in globals that are read, but can't catch them in globals that are assigned to. Because of this, and to discourage the use of globals in general, linter detects cases when a global is only used in one function and can be safely converted to a local variable. Note that in some cases this requires declaring the local variable in the beginning of the function instead of where it's being assigned to.

```
local function testFunc(a)
    if a < 5 then
        -- Global 'b' is only used in the enclosing function; consider changing it to local
        b = 1
    else
        b = 2
    end
    print(b)
end
```

LocalShadow (4)

In Luau, it is valid to shadow locals and globals with a local variable, including doing it in the same function. This can result in subtle bugs, since the shadowing may not be obvious to the reader. This warning detects cases where local variables shadow other local variables in the same function, or global variables used in the script; for more cases of detected shadowing see `LocalShadowPedantic`.

```
local function foo()
    for i=1,10 do
        -- Variable 'i' shadows previous declaration at line 2
        for i=1,10 do
            print(i)
        end
    end
end
```

SameLineStatement (5)

Luau doesn't require the use of semicolons and doesn't automatically insert them at line breaks. When used wisely this results in code that is easy to read and understand, however it can cause subtle issues and hard to understand code when abused by using many different statements on the same line. This warning highlights cases where code should either be broken into multiple lines, or use `;` as a visual guide.

```
-- A new statement is on the same line; add semi-colon on previous statement to silence
if b < 0 then local a = b + 1 print(a, b) end
```

MultiLineStatement (6)

An opposite problem is having statements that span multiple lines. This is good for readability when the code is indented properly, but when it's not it results in code that's hard to understand, as its easy to confuse the next line for a separate statement.

```
-- Statement spans multiple lines; use indentation to silence
print(math.max(1,
math.min(2, 3)))
```

LocalUnused (7)

This warning is one of the few warnings that highlight unused variables. Local variable declarations that aren't used may indicate a bug in the code (for example, there could be a typo in the code that uses the wrong variable) or redundant code that is no longer necessary (for example, calling a function to get its result and never using this result). This warning warns about locals that aren't used; if the locals are not used intentionally they can be prefixed with _ to silence the warning:

```
local x = 1
-- Variable 'y' is never used; prefix with '_' to silence
local y = 2
print(x, x)
```

FunctionUnused (8)

While unused local variables could be useful for debugging, unused functions usually contain dead code that was meant to be removed. Unused functions clutter code, can be a result of typos similar to local variables, and can mislead the reader into thinking that some functionality is supported.

```
-- Function 'bar' is never used; prefix with '_' to silence
local function bar()
end

local function foo()
end

return foo()
```

ImportUnused (9)

In Luau, there's no first-class module system that's part of the syntax, but `require` function acts as an import statement. When a local is initialized with a `require` result, and the local is unused, this is classified as "unused import". Removing unused imports improves code quality because it makes it obvious what the dependencies of the code are:

```
-- Import 'Roact' is never used; prefix with '_' to silence
local Roact = require(game.Packages.Roact)
```

BuiltinGlobalWrite (10)

While the sandboxing model of Luau prevents overwriting built-in globals such as `table` for the entire program, it's still valid to reassign these globals - this results in "global shadowing", where the script's global table contains a custom version of `table` after writing to it. This is problematic because it disables some optimizations, and can result in misleading code. When shadowing built-in globals, use locals instead.

```
-- Built-in global 'math' is overwritten here; consider using a local or changing the name
math = {}
```

PlaceholderRead (11)

_ variable name is commonly used as a placeholder to discard function results. The linter follows this convention and doesn't warn about the use of _ in various cases where a different name would cause a warning otherwise. To make sure the placeholder is only used to write values to it, this warning detects the cases where it's read instead:

```
local _ = 5
-- Placeholder value '_' is read here; consider using a named variable
return _
```

UnreachableCode (12)

In some cases the linter can detect code that is never executed, because all execution paths through the function exit the function or the loop before reaching it. Such code is misleading because it's not always obvious to the reader that it never runs, and as such it should be removed.

```
function cbrt(v)
    if v >= 0 then
        return v ^ (1/3)
    else
        error('cbrt expects a non-negative argument')
    end
    -- Unreachable code (previous statement always returns)
    return 0
end
```

UnknownType (13)

Luau provides several functions to get the value type as a string (`type`, `typeof`), and some Roblox APIs expose class names through string arguments (`Instance.new`). This warning detects incorrect use of the type names by checking the string literals used in type comparisons and function calls.

```
-- Unknown type 'String' (expected primitive type)
if type(v) == "String" then
    print("v is a string")
end
```

ForRange (14)

When using a numeric for, it's possible to make various mistakes when specifying the for bounds. For example, to iterate through the table backwards, it's important to specify the negative step size. This warning detects several cases where the numeric for only runs for 0 or 1 iterations, or when the step doesn't divide the size of the range evenly.

```
-- For loop should iterate backwards; did you forget to specify -1 as step?
for i=#t,1 do
end
```

UnbalancedAssignment (15)

Assignment statements and local variable declarations in Luau support multiple variables on the left side and multiple values on the right side. The number of values doesn't need to match; when the right side has more values, the extra values are discarded, and then the left side has more variables the extra variables are set to `nil`. However, this can result in subtle bugs where a value is omitted mistakenly. This warning warns about cases like this; if the last expression on the right hand side returns multiple values, the warning is not emitted.

```
-- Assigning 2 values to 3 variables initializes extra variables with nil; add 'nil' to value list to silence
local x, y, z = 1, 2
```

ImplicitReturn (16)

In Luau, there's a subtle difference between returning no values from a function and returning `nil`. In many contexts these are equivalent, but when the results are passed to a variadic function (perhaps implicitly), the difference can be observed - for example, `print(foo())` prints nothing if `foo` returns no values, and `nil` if it returns `nil`.

To help write code that has consistent behavior, linter warns about cases when a function implicitly returns no values, if there are cases when it explicitly returns a result. For code like this it's recommended to use explicit `return` or `return nil` at the end of the function (these have different semantics, so the correct version to use depends on the function):

```
local function find(t, expected)
    for k,v in pairs(t) do
        if k == expected then
            return v
        end
    end
    -- Function 'find' can implicitly return no values even though there's an explicit return at line 4; add explicit return to sile
end
```

DuplicateLocal (17)

Luau syntax allows to use the same name for different parameters of a function as well as different local variables declared in the same statement. This is error prone, even if it's occasionally useful, so a warning is emitted in cases like this, unless the duplicate name is the placeholder _:

```
function foo(a, b, a) -- Function parameter 'a' already defined on column 14
end

function obj:method(self) -- Function parameter 'self' already defined implicitly
end

local x, y, x = v:GetComponents() -- Variable 'x' already defined on column 7
```

FormatString (18)

Luau has several library functions that expect a format string that specifies the behavior for the function. These format strings follow a specific syntax that depends on the question; mistakes in these strings can lead to runtime errors or unexpected behavior of the code.

To help make sure that the strings used for these functions are correct, linter checks calls to `string.format`, `string.pack`, `string.packsize`, `string.unpack`, `string.match`, `string.gmatch`, `string.find`, `string.gsub` and `os.date` and issues warnings when the call uses a literal string with an incorrect format:

```
-- Invalid match pattern: invalid capture reference, must refer to a closed capture
local cap = string.match(s, "(%d)%2")

-- Invalid format string: unfinished format specifier
local str = ("%d %"):format(1, 2)
```

Note that with the exception of `string.format` this only works when the function is called via the library, not via the method call (so prefer `string.match(s, "pattern")` to `s:match("pattern")`).

TableLiteral (19)

Table literals are often used in Luau to create objects or specify data. The syntax for table literals is rich but invites mistakes, for example it allows duplicate keys or redundant index specification for values already present in the list form. This warning is emitted for cases where some entries in the table literal are ignored at runtime as they're duplicate:

```
print({
    first = 1,
    second = 2,
    first = 3, -- Table field 'first' is a duplicate; previously defined at line 2
})
```

UninitializedLocal (20)

It's easy to forget to initialize a local variable and then proceed to use it regardless. This can happen due to a typo, or sometimes it can happen because the original variable definition is shadowed. When a local variable doesn't have an initial value and is used without ever being assigned to, this warning is emitted:

```
local foo

if foo then -- Variable 'foo' defined at line 1 is never initialized or assigned; initialize with 'nil' to silence
        print(foo)
end
```

DuplicateFunction (21)

This warning is emitted when a script defines two functions with the same name in the same scope.

The warning is not produced when the functions are defined in different scopes because this is much more likely to be intentional.

```
function foo() end
function foo() end -- Duplicate function definition: 'foo' also defined on line 1

-- OK: the functions are not defined in the same scope.
if x then
    function bar() end
else
    function bar() end
end
```

DeprecatedApi (22)

This warning is emitted when a script accesses a method or field that is marked as deprecated. Use of deprecated APIs is discouraged since they may have performance or correctness issues, may result in unexpected behavior, and could be removed in the future.

```
function getSize(i: Instance)
    return i.DataCost -- Member 'Instance.DataCost' is deprecated
end
```

TableOperations (23)

To manipulate array-like tables, Luau provides a set of functions as part of the standard `table` library. To help use these functions correctly, this warning is emitted when one of these functions is used incorrectly or can be adjusted to be more performant.

```
local t = {}

table.insert(t, 0, 42) -- table.insert uses index 0 but arrays are 1-based; did you mean 1 instead?
table.insert(t, #t+1, 42) -- table.insert will append the value to the table; consider removing the second argument for efficiency
```

In addition, when type information is present, this warning will be emitted when `#` or `ipairs` is used on a table that has no numeric keys or indexers. This helps avoid common bugs like using `#t == 0` to check if a dictionary is empty.

```
local message = { data = { 1, 2, 3 } }

if #message == 0 then -- Using '#' on a table without an array part is likely a bug
end
```

DuplicateCondition (24)

When checking multiple conditions via `and/or` or `if/elseif`, a copy & paste error may result in checking the same condition redundantly. This almost always indicates a bug, so a warning is emitted when use of a duplicate condition is detected.

```
assert(self._adorns[normID1] and self._adorns[normID1]) -- Condition has already been checked on column 8
```

MisleadingAndOr (25)

In Lua, there is no first-class ternary operator but it can be emulated via `a and b or c` pattern. However, due to how boolean evaluation works, if `b` is `false` or `nil`, the resulting expression evaluates to `c` regardless of the value of `a`. Luau solves this problem with the `if a then b else c` expression; a warning is emitted for and-or expressions where the first alternative is `false` or `nil` because it's almost always a bug.

```
-- The and-or expression always evaluates to the second alternative because the first alternative is false; consider using if-then-el
local x = flag and false or true
```

The code above can be rewritten as follows to avoid the warning and the associated bug:

```
local x = if flag then false else true
```

CommentDirective (26)

Luau uses comments that start from `!` to control certain aspects of analysis, for example setting type checking mode via `--!strict` or disabling individual lints with `--!nolint`. Unknown directives are ignored, for example `--!nostrict` doesn't have any effect on the type checking process as the correct spelling is `--!nonstrict`. This warning flags comment directives that are ignored during processing:

```
--!nostrict
-- Unknown comment directive 'nostrict'; did you mean 'nonstrict'?"
```

IntegerParsing (27)

Luau parses hexadecimal and binary literals as 64-bit integers before converting them to Luau numbers. As a result, numbers that exceed $2^{64 \text{ are silently truncated to } 2}64$, which can result in unexpected program behavior. This warning flags literals that are truncated:

```
-- Hexadecimal number literal exceeded available precision and was truncated to 2^64
local x = 0x1111111111111111111111111111111111111
```

Luau numbers are represented as double precision IEEE754 floating point numbers; they can represent integers up to 2^53 exactly, but larger integers may lose precision. This warning also flags literals that are parsed with a precision loss:

```
-- Number literal exceeded available precision and was truncated to closest representable number
local x = 9007199254740993
```

ComparisonPrecedence (28)

Because of operator precedence rules, not X == Y parses as (not X) == Y; however, often the intent was to invert the result of the comparison. This warning flags erroneous conditions like that, as well as flagging cases where two comparisons happen in a row without any parentheses:

```
-- not X == Y is equivalent to (not X) == Y; consider using X ~= Y, or wrap one of the expressions in parentheses to silence
if not x == 5 then
end

-- X <= Y <= Z is equivalent to (X <= Y) <= Z; wrap one of the expressions in parentheses to silence
if 1 <= x <= 3 then
end
```

---

title: "Luau News" permalink: /news/ layout: posts sidebar: "none"

---

permalink: /performance title: Performance toc: true

One of main goals of Luau is to enable high performance code, with gameplay code being the main use case. This can be viewed as two separate goals:

- Make idiomatic code that wasn't tuned faster
- Enable even higher performance through careful tuning

Both of these goals are important - it's insufficient to just focus on the highly tuned code, and all things being equal we prefer to raise all boats by implementing general optimizations. However, in some cases it's important to be aware of optimizations that Luau does and doesn't do.

Worth noting is that Luau is focused on, first and foremost, stable high performance code in interpreted context. This is because JIT compilation is not available on many platforms Luau runs on, and AOT compilation would only work for code that Roblox ships (and even that does not always work). This is in stark contrast with LuaJIT that, while providing an excellent interpreter as well, focuses a lot of the attention on JIT (with many optimizations unavailable in the interpreter).

Luau eventually plans to implement JIT on some platforms, but this is subject to careful memory safety analysis and is likely to not be deployed for client-side scripts, as the extra risk involved in JITs is much more pronounced when it may affect players.

The rest of this document goes into some optimizations that Luau employs and how to best leverage them when writing code. The document is not complete - a lot of optimizations are transparent to the user and involve detailed low-level tuning of various parts that is not described here - and all of this is subject to change without notice, as it doesn't affect the semantics of valid code.

Fast bytecode interpreter

Luau features a very highly tuned portable bytecode interpreter. It's similar to Lua interpreter in that it's written in C, but it's highly tuned to yield efficient assembly when compiled with Clang and latest versions of MSVC. On some workloads it can match the performance of LuaJIT interpreter which is written in highly specialized assembly. We are continuing to tune the interpreter and the bytecode format over time; while extra performance can be extracted by rewriting the interpreter in assembly, we're unlikely to ever do that as the extra gains at this point are marginal, and we gain a lot from C in terms of portability and being able to quickly implement new optimizations.

Of course the interpreter isn't typical C code - it uses many tricks to achieve extreme levels of performance and to coerce the compiler to produce efficient assembly. Due to a better bytecode design and more efficient dispatch loop it's noticeably faster than Lua 5.x (including Lua 5.4 which made some of the changes similar to Luau, but doesn't come close). The bytecode design was partially inspired by excellent LuaJIT interpreter. Most computationally intensive scripts only use the interpreter core loop and builtins, which on x64 compiles into ~16 KB, thus leaving half of the instruction cache for other infrequently called code.

Optimizing compiler

Unlike Lua and LuaJIT, Luau uses a multi-pass compiler with a frontend that parses source into an AST and a backend that generates bytecode from it. This carries a small penalty in terms of compilation time, but results in more flexible code and, crucially, makes it easier to optimize the generated bytecode.

> *Note: Compilation throughput isn't the main focus in Luau, but our compiler is reasonably fast; with all currently implemented optimizations enabled, it compiles 950K lines of Luau code in 1 second on a single core of a desktop Ryzen 5900X CPU, producing bytecode and debug information.*

While bytecode optimizations are limited due to the flexibility of Luau code (e.g. `a * 1` may not be equivalent to `a` if `*` is overloaded through metatables), even in absence of type information Luau compiler can perform some optimizations such as "deep" constant folding across functions and local variables, perform upvalue optimizations for upvalues that aren't mutated, do analysis of builtin function usage, optimize the instruction sequences for multiple variable assignments, and some peephole optimizations on the resulting bytecode. The compiler can also be instructed to use more aggressive optimizations by enabling optimization level 2 (`-O2` in CLI tools), some of which are documented further on this page.

Most bytecode optimizations are performed on individual statements or functions, however the compiler also does a limited amount of interprocedural optimizations; notably, calls to local functions can be optimized with the knowledge of the argument count or number of return values involved. Interprocedural optimizations are limited to a single module due to the compilation model.

Luau compiler currently doesn't use type information to do further optimizations, however early experiments suggest that we can extract further wins. Because we control the entire stack (unlike e.g. TypeScript where the type information is discarded completely before reaching the VM), we have more flexibility there and can make some tradeoffs during codegen even if the type system isn't completely sound. For example, it might be reasonable to assume that in presence of known types, we can infer absence of side effects for arithmetic operations and builtins - if the runtime types mismatch due to intentional violation of the type safety through global injection, the code will still be safely sandboxed; this may unlock optimizations such as common subexpression elimination and allocation hoisting without a JIT. This is speculative pending further research.

Epsilon-overhead debugger

It's important for Luau to have stable and predictable performance. Something that comes up in Lua-based environments often is the use of line hooks to implement debugging (both for breakpoints and for stepping). This is problematic because the support for hooks is typically not free in general, but importantly once the hook is enabled, calling the hook has a considerable overhead, and the hook itself may be very costly to evaluate since it will need to associate the script:line pair with the breakpoint information.

Luau does not support hooks at all, and relies on first-class support for breakpoints (using bytecode patching) and single-stepping (using a custom interpreter loop) to implement debugging. As a result, the presence of breakpoints doesn't slow the script execution down - the only noticeable discrepancy between running code under a debugger and without a debugger should be in cases where breakpoints are evaluated and skipped based on breakpoint conditions, or when stepping over long-running fragments of code.

Inline caching for table and global access

Table access for field lookup is optimized in Luau using a mechanism that blends inline caching (classically used in Java/JavaScript VMs) and HREFs (implemented in LuaJIT). Compiler can predict the hash slot used by field lookup, and the VM can correct this prediction dynamically.

As a result, field access can be very fast in Luau, provided that:

- The field name is known at compile time. To make sure this is the case, `table.field` notation is recommended, although the compiler will also optimize `table["field"]` when the expression is known to be a constant string.
- The field access doesn't use metatables. The fastest way to work with tables in Luau is to store fields directly inside the table, and store methods in the metatable (see below); access to "static" fields in classic OOP designs is best done through `Class.StaticField` instead of `object.StaticField`.
- The object structure is usually uniform. While it's possible to use the same function to access tables of different shape - e.g. `function getX(obj) return obj.x end` can be used on any table that has a field `"x"` - it's best to not vary the keys used in the tables too much, as it defeats this optimization.

The same optimization is applied to the custom globals declared in the script, although it's best to avoid these altogether by using locals instead. Still, this means that the difference between `function` and `local function` is less pronounced in Luau.

Importing global access chains

While global access for library functions can be optimized in a similar way, this optimization breaks down when the global table is using sandboxing through metatables, and even when globals aren't sandboxed, `math.max` still requires two table accesses.

It's always possible to "localize" the global accesses by using `local max = math.max`, but this is cumbersome - in practice it's easy to forget to apply this optimization. To avoid relying on programmers remembering to do this, Luau implements a special optimization called "imports", where most global chains such as `math.max` are resolved when the script is loaded instead of when the script is executed.

This optimization relies on being able to predict the shape of the environment table for a given function; this is possible due to global sandboxing, however this optimization is invalid in some cases:

- `loadstring` can load additional code that runs in context of the caller's environment
- `getfenv`/`setfenv` can directly modify the environment of any function

The use of any of these functions performs a dynamic deoptimization, marking the affected environment as "impure". The optimizations are only in effect on functions with "pure" environments - because of this, the use of `loadstring`/`getfenv`/`setfenv` is not recommended. Note that `getfenv` deoptimizes the environment even if it's only used to read values from the environment.

> *Note: Luau still supports these functions as part of our backwards compatibility promise, although we'd love to switch to Lua 5.2's `_ENV` as that mechanism is cleaner and doesn't require costly dynamic deoptimization.*

Fast method calls

Luau specializes method calls to improve their performance through a combination of compiler, VM and binding optimizations. Compiler emits a specialized instruction sequence when methods are called through `obj:Method` syntax (while this isn't idiomatic anyway, you should avoid `obj.Method(obj)`). When the object in question is a Lua table, VM performs some voodoo magic based on inline caching to try to quickly discover the implementation of this method through the metatable.

For this to be effective, it's crucial that `__index` in a metatable points to a table directly. For performance reasons it's strongly recommended to avoid `__index` functions as well as deep `__index` chains; an ideal object in Luau is a table with a metatable that points to itself through `__index`.

When the object in question is a reflected userdata, a special mechanism called "namecall" is used to minimize the interop cost. In classical Lua binding model, `obj:Method` is called in two steps, retrieving the function object (`obj.Method`) and calling it; both steps are often implemented in C++, and the method retrieval needs to use a method object cache - all of this makes method calls slow.

Luau can directly call the method by name using the "namecall" extension, and an optimized reflection layer can retrieve the correct method quickly through more voodoo magic based on string interning and custom Luau features that aren't exposed through Luau scripts.

As a result of both optimizations, common Lua tricks of caching the method in a local variable aren't very productive in Luau and aren't recommended either.

Specialized builtin function calls

Due to global sandboxing and the ability to dynamically deoptimize code running in impure environments, in pure environments we go beyond optimizing the interpreter and optimize many built-in functions through a "fastcall" mechanism.

For this mechanism to work, function call must be "obvious" to the compiler - it needs to call a builtin function directly, e.g. `math.max(x, 1)`, although it also works if the function is "localized" (`local max = math.max`); this mechanism doesn't work for indirect function calls unless they were inlined during compilation, and doesn't work for method calls (so calling `string.byte` is more efficient than `s:byte`).

The mechanism works by directly invoking a highly specialized and optimized implementation of a builtin function from the interpreter core loop without setting up a stack frame and omitting other work; additionally, some fastcall specializations are partial in that they don't support all types of arguments, for example all `math` library builtins are only specialized for numeric arguments, so calling `math.abs` with a string argument will fall back to the slower implementation that will do string->number coercion.

As a result, builtin calls are very fast in Luau - they are still slightly slower than core instructions such as arithmetic operations, but only slightly so. The set of fastcall builtins is slowly expanding over time and as of this writing contains `assert`, `type`, `typeof`, `rawget`/`rawset`/`rawequal`, `getmetatable`/`setmetatable`, `tonumber`/`tostring`, all functions from `math` (except `noise` and `random`/`randomseed`) and `bit32`, and some functions from `string` and `table` library.

Some builtin functions have partial specializations that reduce the cost of the common case further. Notably:

- `assert` is specialized for cases when the assertion return value is not used and the condition is truthy; this helps reduce the runtime cost of assertions to the extent possible
- `bit32.extract` is optimized further when field and width selectors are constant
- `select` is optimized when the second argument is `...`; in particular, `select(x, ...)` is O(1) when using the builtin dispatch mechanism even though it's normally O(N) in variadic argument count.

Some functions from `math` library like `math.floor` can additionally take advantage of advanced SIMD instruction sets like SSE4.1 when available.

In addition to runtime optimizations for builtin calls, many builtin calls, as well as constants like `math.pi`/`math.huge`, can also be constant-folded by the bytecode compiler when using aggressive optimizations (level 2); this currently applies to most builtin calls with constant arguments and a single return value. For builtin calls that can not be constant folded, compiler assumes knowledge of argument/return count (level 2) to produce more efficient bytecode instructions.

Optimized table iteration

Luau implements a fully generic iteration protocol; however, for iteration through tables in addition to generalized iteration (`for .. in t`) it recognizes three common idioms (`for .. in ipairs(t)`, `for .. in pairs(t)` and `for .. in next, t`) and emits specialized bytecode that is carefully optimized using custom internal iterators.

As a result, iteration through tables typically doesn't result in function calls for every iteration; the performance of iteration using generalized iteration, `pairs` and `ipairs` is comparable, so generalized iteration (without the use of `pairs`/`ipairs`) is recommended unless the code needs to be compatible with vanilla Lua or the specific semantics of `ipairs` (which stops at the first `nil` element) is required. Additionally, using generalized iteration avoids calling `pairs` when the loop starts which can be noticeable when the table is very short.

Iterating through array-like tables using `for i=1,#t` tends to be slightly slower because of extra cost incurred when reading elements from the table.

Optimized table length

Luau tables use a hybrid array/hash storage, like in Lua; in some sense "arrays" don't truly exist and are an internal optimization, but some operations, notably `#t` and functions that depend on it, like `table.insert`, are defined by the Luau/Lua language to allow internal optimizations. Luau takes advantage of that fact.

Unlike Lua, Luau guarantees that the element at index `#t` is stored in the array part of the table. This can accelerate various table operations that use indices limited by `#t`, and this makes `#t` worst-case complexity O(logN), unlike Lua where the worst case complexity is O(N). This also accelerates computation of this value for small tables like `{ [1] = 1 }` since we never need to look at the hash part.

The "default" implementation of `#t` in both Lua and Luau is a binary search. Luau uses a special branch-free (depending on the compiler...) implementation of the binary search which results in 50+% faster computation of table length when it needs to be computed from scratch.

Additionally, Luau can cache the length of the table and adjust it following operations like `table.insert`/`table.remove`; this means that in practice, `#t` is almost always a constant time operation.

Creating and modifying tables

Luau implements several optimizations for table creation. When creating object-like tables, it's recommended to use table literals (`{ ... }`) and to specify all table fields in the literal in one go instead of assigning fields later; this triggers an optimization inspired by LuaJIT's "table templates" and results in higher performance when creating objects. When creating array-like tables, if the maximum size of the table is known up front, it's recommended to use `table.create` function which can create an empty table with preallocated storage, and optionally fill it with a given value.

When the exact table shape isn't known, Luau compiler can still predict the table capacity required in case the table is initialized with an empty literal (`{}`) and filled with fields subsequently. For example, the following code creates a correctly sized table implicitly:

```
local v = {}
v.x = 1
v.y = 2
v.z = 3
return v
```

When appending elements to tables, it's recommended to use `table.insert` (which is the fastest method to append an element to a table if the table size is not known). In cases when a table is filled sequentially, however, it can be more efficient to use a known index for insertion - together with preallocating tables using `table.create` this can result in much faster code, for example this is the fastest way to build a table of squares:

```
local t = table.create(N)

for i=1,N do
	t[i] = i * i
end
```

Native vector math

Luau uses tagged value storage - each value contains a type tag and the data that represents the value of a given type. Because of the need to store 64-bit double precision numbers *and* 64-bit pointers, we don't use NaN tagging and have to pay the cost of 16 bytes per value.

We take advantage of this to provide a native value type that can store a 32-bit floating point vector with 3 components. This type is fundamental to game computations and as such it's important to optimize the storage and the operations with that type - our VM implements first class support for all math operations and component manipulation, which essentially means we have native 3-wide SIMD support. For code that uses many vector values this results in significantly smaller GC pressure and significantly faster execution, and gives programmers a mechanism to hand-vectorize numeric code if need be.

Optimized upvalue storage

Lua implements upvalues as garbage collected objects that can point directly at the thread's stack or, when the value leaves the stack frame (and is "closed"), store the value inside the object. This representation is necessary when upvalues are mutated, but inefficient when they aren't - and 90% or more of upvalues aren't mutated in typical Lua code. Luau takes advantage of this by reworking upvalue storage to prioritize immutable upvalues - capturing upvalues that don't change doesn't require extra allocations or upvalue closing, resulting in faster closure allocation, faster execution, faster garbage collection and faster upvalue access due to better memory locality.

Note that "immutable" in this case only refers to the variable itself - if the variable isn't assigned to it can be captured by value, even if it's a table that has its contents change.

When upvalues are mutable, they do require an extra allocated object; we carefully optimize the memory consumption and access cost for mutable upvalues to reduce the associated overhead.

Closure caching

With optimized upvalue storage, creating new closures (function objects) is more efficient but still requires allocating a new object every time. This can be problematic for cases when functions are passed to algorithms like `table.sort` or functions like `pcall`, as it results in excessive allocation traffic which then leads to more work for garbage collector.

To make closure creation cheaper, Luau compiler implements closure caching - when multiple executions of the same function expression are guaranteed to result in the function object that is semantically identical, the compiler may cache the closure and always return the same object. This changes the function identity which may affect code that uses function objects as table keys, but preserves the calling semantics - compiler will only do this if calling the original (cached) function behaves the same way as a newly created function would. The heuristics used for this optimization are subject to change; currently, the compiler will cache closures that have no upvalues, or all upvalues are immutable (see previous section) and are declared at the module scope, as the module scope is (almost always) evaluated only once.

Fast memory allocator

Similarly to LuaJIT, but unlike vanilla Lua, Luau implements a custom allocator that is highly specialized and tuned to the common allocation workloads we see. The allocator design is inspired by classic pool allocators as well as the excellent `mimalloc`, but through careful domain-specific tuning it beats all general purpose allocators we've tested, including `rpmalloc`, `mimalloc`, `jemalloc`, `ptmalloc` and `tcmalloc`.

This doesn't mean that memory allocation in Luau is free - it's carefully optimized, but it still carries a cost, and a high rate of allocations requires more work from the garbage collector. The garbage collector is incremental, so short of some edge cases this rarely results in visible GC pauses, but can impact the throughput since scripts will interrupt to perform "GC assists" (helping clean up the garbage). Thus for high performance Luau code it's recommended to avoid allocating memory in tight loops, by avoiding temporary table and userdata creation.

In addition to a fast allocator, all frequently used structures in Luau have been optimized for memory consumption, especially on 64-bit platforms, compared to Lua 5.1 baseline. This helps to reduce heap memory footprint and improve performance in some cases by reducing the memory bandwidth impact of garbage collection.

Optimized libraries

While the best performing code in Luau spends most of the time in the interpreter, performance of the standard library functions is critical to some applications. In addition to specializing many small and simple functions using the builtin call mechanism, we spend extra care on optimizing all library functions and providing additional functions beyond the Lua standard library that help achieve good performance with idiomatic code.

Functions from the `table` library like `insert`, `remove` and `move` have been tuned for performance on array-like tables, achieving 3x and more performance compared to un-tuned versions, and Luau provides additional functions like `table.create` and `table.find` to achieve further speedup when applicable. Our implementation of `table.sort` is using `introsort` algorithm which results in guaranteed worst case `NlogN` complexity regardless of the input, and, together with the array-like specializations, helps achieve ~4x speedup on average.

For `string` library, we use a carefully tuned dynamic string buffer implementation; it is optimized for smaller strings to reduce garbage created during string manipulation, and for larger strings it allows to produce a large string without extra copies, especially in cases where the resulting size is known ahead of time. Additionally, functions like `format` have been tuned to avoid the overhead of `sprintf` where possible, resulting in further speedups.

Improved garbage collector pacing

Luau uses an incremental garbage collector which does a little bit of work every so often, and at no point does it stop the world to traverse the entire heap. The runtime will make sure that the collector runs interspersed with the program execution as the program allocates additional memory, which is known as "garbage collection assists", and can also run in response to explicit garbage collection invocation via `lua_gc`. In interactive environments such as video game engines it's possible, and even desirable, to request garbage collection every frame to make sure assists are minimized, since that allows scheduling the garbage collection to run concurrently with other engine processing that doesn't involve script execution.

Inspired by excellent work by Austin Clements on Go's garbage collector pacer, we've implemented a pacing algorithm that uses a proportional–integral–derivative controller to estimate internal garbage collector tunables to reach a target heap size, defined as a percentage of the live heap data (which is more intuitive and actionable than Lua 5.x "GC pause" setting). Luau runtime also estimates the allocation rate making it easy (given uniform allocation rates) to adjust the per-frame garbage collection requests to do most of the required GC work outside of script execution.

Reduced garbage collector pauses

While Luau uses an incremental garbage collector, once per each collector cycle it runs a so-called "atomic" step. While all other GC steps can do very little work by only looking at a few objects at a given time, which means that the collector can have arbitrarily short pauses, the "atomic" step needs to traverse some amount of data that, in some cases, may scale with the application heap. Since atomic step is indivisible, it can result in occasional pauses on the order of tens of milliseconds, which is problematic for interactive applications. We've implemented a series of optimizations to help reduce the atomic step.

Normally objects that have been modified after the GC marked them in an incremental mark phase need to be rescanned during atomic phase, so frequent modifications of existing tables may result in a slow atomic step. To address this, we run a "remark" step where we traverse objects that have been modified after being marked once more (incrementally); additionally, the write barrier that triggers for object modifications changes the transition logic during remark phase to reduce the probability that the object will need to be rescanned.

Another source of scalability challenges is coroutines. Writes to coroutine stacks don't use a write barrier, since that's prohibitively expensive as they are too frequent. This means that coroutine stacks need to be traversed during atomic step, so applications with many coroutines suffer large atomic pauses. To address this, we implement incremental marking of coroutines: marking a coroutine makes it "inactive" and resuming a coroutine (or pushing extra objects on the coroutine stack via C API) makes it "active". Atomic step only needs to traverse active coroutines again, which reduces the cost of atomic step by effectively making coroutine collection incremental as well.

While large tables can be a problem for incremental GC in general since currently marking a single object is indivisible, large weak tables are a unique challenge because they also need to be processed during atomic phase, and the main use case for weak tables - object caches - may result in tables with large capacity but few live objects in long-running applications that exhibit bursts of activity. To address this, weak tables in Luau can be marked as "shrinkable" by including `s` as part of `__mode` string, which results in weak tables being resized to the optimal capacity during GC. This option may result in missing keys during table iteration if the table is resized while iteration is in progress and as such is only recommended for use in specific circumstances.

Optimized garbage collector sweeping

The incremental garbage collector in Luau runs three phases for each cycle: mark, atomic and sweep. Mark incrementally traverses all live objects, atomic finishes various operations that need to happen without mutator intervention (see previous section), and sweep traverses all objects in the heap, reclaiming memory used by dead objects and performing minor fixup for live objects. While objects allocated during the mark phase are traversed in the same cycle and thus may get reclaimed, objects allocated during the sweep phase are considered live. Because of this, the faster the sweep phase completes, the less garbage will accumulate; and, of course, the less time sweeping takes the less overhead there is from this phase of garbage collection on the process.

Since sweeping traverses the whole heap, we maximize the efficiency of this traversal by allocating garbage-collected objects of the same size in 16 KB pages, and traversing each page at a time, which is otherwise known as a paged sweeper. This ensures good locality of reference as consecutively swept objects are contiugous in memory, and allows us to spend no memory for each object on sweep-related data or allocation metadata, since paged sweeper doesn't need to be able to free objects without knowing which page they are in. Compared to linked list based sweeping that Lua/LuaJIT implement, paged sweeper is 2-3x faster, and saves 16 bytes per object on 64-bit platforms.

Function inlining and loop unrolling

By default, the bytecode compiler performs a series of optimizations that result in faster execution of the code, but they preserve both execution semantics and debuggability. For example, a function call is compiled as a function call, which may be observable via `debug.traceback`; a loop is compiled as a loop, which may be observable via `lua_getlocal`. To help improve performance in cases where these restrictions can be relaxed, the bytecode compiler implements additional optimizations when optimization level 2 is enabled (which requires using `-O2` switch when using Luau CLI), namely function inlining and loop unrolling.

Only loops with loop bounds known at compile time, such as `for i=1,4 do`, can be unrolled. The loop body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if unrolling should be performed.

Only local functions (defined either as `local function foo` or `local foo = function`) can be inlined. The function body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if each call to the function should be inlined instead. Additionally recursive invocations of a function can’t be inlined at this time, and inlining is completely disabled for modules that use `getfenv`/`setfenv` functions.

In both cases, in addition to removing the overhead associated with function calls or loop iteration, these optimizations can additionally benefit by enabling additional optimizations, such as constant folding of expressions dependent on loop iteration variable or constant function arguments, or using more efficient instructions for certain expressions when the inputs to these instructions are constants.

---

permalink: /profile title: Profiling toc: true

One of main goals of Luau is to enable high performance code. To help with that goal, we are relentlessly optimizing the compiler and runtime - but ultimately, performance of their code is in developers' hands, and is a combination of good algorithm design and implementation that adheres to the strengths of the language. To help write efficient code, Luau provides a built-in profiler that samples the execution of the program and outputs a profiler dump that can be converted to an interactive flamegraph.

To run the profiler, make sure you have an optimized build of the interpreter (otherwise profiling results are going to be very skewed) and run it with `--profile` argument:

```
$ luau --profile tests/chess.lua
OK      8902    rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
OK      2039    r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 0
OK      2812    8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - 0 0
OK      9467    r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1
OK      1486    rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8
OK      2079    r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10
Profiler dump written to profile.out (total runtime 2.034 seconds, 20344 samples, 374 stacks)
GC: 0.378 seconds (18.58%), mark 46.80%, remark 3.33%, atomic 1.93%, sweepstring 6.77%, sweep 41.16%
```

The resulting `profile.out` file can be converted to an SVG file by running `perfgraph.py` script that is part of Luau repository:

```
$ python tools/perfgraph.py profile.out >profile.svg
```

This produces an SVG file that can be opened in a browser (the image below is clickable):

profile.svg (/assets/images/chess-profile.svg)

In a flame graph visualization, the individual bars represent function calls, the width represents how much of the total program runtime they execute, and the nesting matches the call stack encountered during program execution. This is a fantastic visualization technique that allows you to hone in on the specific bottlenecks affecting your program performance, optimize those exact bottlenecks, and then re-generate the profile data and visualizer, and look for the next set of true bottlenecks (if any).

Hovering your mouse cursor over individual sections will display detailed function information in the status bar and in a tooltip. If you want to Search for a specific named function, use the Search field in the upper right, or press Ctrl+F.

Notice that some of the bars in the screenshot don't have any text. In some cases, there isn't enough room in the size of the bar to display the name. You can hover your mouse over those bars to see the name and source location of the function in the tool tip, or double-click to zoom in on that part of the flame graph.

Some tooltips will have a source location for the function you're hovering over, but no name. Those are anonymous functions, or functions that were not declared in a way that allows Luau compiler to track the name. To fill in more names, you may want to make these changes to your code:

`local myFunc = function() --[[ work ]] end` -> `local function myFunc() --[[ work ]] end`

Even without these changes, you can hover over a given bar with no visible name and see it's source location.

As any sampling profiler, this profiler relies on gathering enough information for the resulting output to be statistically meaningful. It may miss short functions if they aren't called often enough. By default the profiler runs at 10 kHz, this can be customized by passing a different parameter to `--profile=`. Note that higher frequencies result in higher profiling overhead and longer program execution, potentially skewing the results.

This profiler doesn't track leaf C functions and instead attributes the time spent there to calling Luau functions. As a result, when thinking about why a given function is slow, consider not just the work it does immediately but also the library functions it calls.

This profiler tracks time consumed by Luau thread stacks; when a thread calls another thread via `coroutine.resume`, the time spent is not attributed to the parent thread that's waiting for resume results. This limitation will be removed in the future.

---

permalink: /sandbox title: Sandboxing toc: true

Luau is safe to embed. Broadly speaking, this means that even in the face of untrusted (and in Roblox case, actively malicious) code, the language and the standard library don't allow unsafe access to the underlying system, and don't have known bugs that allow escaping out of the sandbox (e.g. to gain native code execution through ROP gadgets et al). Additionally, the VM provides extra features to implement isolation of privileged code from unprivileged code and protect one from the other; this is important if the embedding environment decides to expose some APIs that may not be safe to call from untrusted code, for example because they do provide controlled access to the underlying system or risk PII exposure through fingerprinting etc.

This safety is achieved through a combination of removing features from the standard library that are unsafe, adding features to the VM that make it possible to implement sandboxing and isolation, and making sure the implementation is safe from memory safety issues using fuzzing.

Of course, since the entire stack is implemented in C++, the sandboxing isn't formally proven - in theory, compiler or the standard library can have exploitable vulnerabilities. In practice these are very rare and usually found and fixed quickly. While implementing the stack in a safer language such as Rust would make it easier to provide these guarantees, to our knowledge (based on prior art) this would make it difficult to reach the level of performance required.

Library

Parts of the Lua 5.x standard library are unsafe. Some of the functions provide access to the host operating system, including process execution and file reads. Some functions lack sufficient memory safety checks. Some functions are safe if all code is untrusted, but can break the isolation barrier between trusted and untrusted code.

The following libraries and global functions have been removed as a result:

- `io.` library has been removed entirely, as it gives access to files and allows running processes
- `package.` library has been removed entirely, as it gives access to files and allows loading native modules
- `os.` library has been cleaned up from file and environment access functions (`execute`, `exit`, etc.). The only supported functions in the library are `clock`, `date`, `difftime` and `time`.
- `debug.` library has been removed to a large extent, as it has functions that aren't memory safe and other functions break isolation; the only supported functions are `traceback` and `info` (which is similar to `debug.getinfo` but has a slightly different interface).
- `dofile` and `loadfile` allowed access to file system and have been removed.

To achieve memory safety, access to function bytecode has been removed. Bytecode is hard to validate and using untrusted bytecode may lead to exploits. Thus, `loadstring` doesn't work with bytecode inputs, and `string.dump`/`load` have been removed as they aren't necessary anymore. When embedding Luau, bytecode should be encrypted/signed to prevent MITM attacks as well, as the VM assumes that the bytecode was generated by the Luau compiler (which never produces invalid/unsafe bytecode).

Finally, to make isolation possible within the same VM, the following global functions have reduced functionality:

- `collectgarbage` only works with `"count"` argument, as modifying the state of GC can interfere with the expectations of other code running in the process. As such, `collectgarbage()` became an inferior version of `gcinfo()` and is deprecated.
- `newproxy` only works with `true`/`false`/`nil` arguments.
- `module` allowed overriding global packages and was removed as a result.

> *Note: `getfenv`/`setfenv` result in additional isolation challenges, as they allow injecting globals into scripts on the call stack. Ideally, these should be disabled as well, but unfortunately Roblox community relies on these for various reasons. This can be mitigated by limiting interaction between trusted and untrusted code, and/or using separate VMs.*

Environment

The modification to the library functions are sufficient to make embedding safe, but aren't sufficient to provide isolation within the same VM. It should be noted that to achieve guaranteed isolation, it's advisable to load trusted and untrusted code into separate VMs; however, even within the same VM Luau provides additional safety features to make isolation cheaper.

When initializing the default globals table, the tables are protected from modification:

- All libraries (`string`, `math`, etc.) are marked as readonly
- The string metatable is marked as readonly
- The global table itself is marked as readonly

This is using the VM feature that is not accessible from scripts, that prevents all writes to the table, including assignments, `rawset` and `setmetatable`. This makes sure that globals can't be monkey-patched in place, and can only be substituted through `setfenv`.

By itself this would mean that code that runs in Luau can't use globals at all, since assigning globals would fail. While this is feasible, in Roblox we solve this by creating a new global table for each script, that uses `__index` to point to the builtin global table. This safely sandboxes the builtin globals while still allowing writing globals from each script. This also means that short of exposing special shared globals from the host, all scripts are isolated from each other.

`__gc`

Lua 5.1 exposes a `__gc` metamethod for userdata, which can be used on proxies (`newproxy`) to hook into garbage collector. Later versions of Lua extend this mechanism to work on tables.

This mechanism is bad for performance, memory safety and isolation:

- In Lua 5.1, `__gc` support requires traversing userdata lists redundantly during garbage collection to filter out finalizable objects
- In later versions of Lua, userdata that implement `__gc` are split into separate lists; however, finalization prolongs the lifetime of the finalized objects which results in less prompt memory reclamation, and two-step destruction results in extra cache misses for userdata
- `__gc` runs during garbage collection in context of an arbitrary thread which makes the thread identity mechanism used in Roblox to support trusted Luau code invalid

- Objects can be removed from weak tables *after* being finalized, which means that accessing these objects can result in memory safety bugs, unless all exposed userdata methods guard against use-after-gc.
- If `__gc` method ever leaks to scripts, they can call it directly on an object and use any method exposed by that object after that. This means that `__gc` and all other exposed methods must support memory safety when called on a destroyed object.

Because of these issues, Luau does not support `__gc`. Instead it uses tag-based destructors that can perform additional memory cleanup during userdata destruction; crucially, these are only available to the host (so they can never be invoked manually), and they run right before freeing the userdata memory block which is both optimal for performance, and guaranteed to be memory safe.

For monitoring garbage collector behavior the recommendation is to use weak tables instead.

Interrupts

In addition to preventing API access, it can be important for isolation to limit the memory and CPU usage of code that runs inside the VM.

By default, no memory limits are imposed on the running code, so it's possible to exhaust the address space of the host; this is easy to configure from the host for Luau allocations, but of course with a rich API surface exposed by the host it's hard to eliminate this as a possibility. Memory exhaustion doesn't result in memory safety issues or any particular risk to the system that's running the host process, other than the host process getting terminated by the OS.

Limiting CPU usage can be equally challenging with a rich API. However, Luau does provide a VM-level feature to try to contain runaway scripts which makes it possible to terminate any script externally. This works through a global interrupt mechanism, where the host can setup an interrupt handler at any point, and any Luau code is guaranteed to call this handler "eventually" (in practice this can happen at any function call or at any loop iteration). This still leaves the possibility of a very long running script open if the script manages to find a way to call a single C function that takes a lot of time, but short of that the interruption is very prompt.

Roblox sets up the interrupt handler using a watchdog that:

- Limits the runtime of any script in Studio to 10 seconds (configurable through Studio settings)
- Upon client shutdown, interrupts execution of every running script 1 second after shutdown

---

permalink: /syntax title: Syntax toc: true

Luau uses the baseline syntax of Lua 5.1 (https://www.lua.org/manual/5.1/manual.html#2). For detailed documentation, please refer to the Lua manual, this is an example:

```
local function tree_insert(tree, x)
    local lower, equal, greater = split(tree.root, x)
    if not equal then
        equal = {
            x = x,
            y = math.random(0, 2^31-1),
            left = nil,
            right = nil
        }
    end
    tree.root = merge3(lower, equal, greater)
end
```

Note that future versions of Lua extend the Lua 5.1 syntax with more features; Luau does support string literal extensions but does not support other 5.x additions; for details please refer to compatibility section (compatibility).

The rest of this document documents additional syntax used in Luau.

String literals

Luau implements support for hexadecimal (`\x`), Unicode (`\u`) and `\z` escapes for string literals. This syntax follows Lua 5.3 syntax (https://www.lua.org/manual/5.3/manual.html#3.1):

- `\xAB` inserts a character with the code 0xAB into the string
- `\u{ABC}` inserts a UTF8 byte sequence that encodes U+0ABC character into the string (note that braces are mandatory)
- `\z` at the end of the line inside a string literal ignores all following whitespace including newlines, which can be helpful for breaking long literals into multiple lines.

Number literals

In addition to basic integer and floating-point decimal numbers, Luau supports:

- Hexadecimal integer literals, `0xABC` or `0XABC`
- Binary integer literals, `0b01010101` or `0B01010101`

- Decimal separators in all integer literals, using _ for readability: `1_048_576`, `0xFFFF_FFFF`, `0b_0101_0101`

Note that Luau only has a single number type, a 64-bit IEEE754 double precision number (which can represent integers up to 2^53 exactly), and larger integer literals are stored with precision loss.

Continue statement

In addition to `break` in all loops, Luau supports `continue` statement. Similar to `break`, `continue` must be the last statement in the block.

Note that unlike `break`, `continue` is not a keyword. This is required to preserve backwards compatibility with existing code; so this is a `continue` statement:

```
if x < 0 then
    continue
end
```

Whereas this is a function call:

```
if x < 0 then
    continue()
end
```

When used in `repeat..until` loops, `continue` can not skip the declaration of a local variable if that local variable is used in the loop condition; code like this is invalid and won't compile:

```
repeat
    do continue end
    local a = 5
until a > 0
```

Compound assignments

Luau supports compound assignments with the following operators: `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `..=`. Just like regular assignments, compound assignments are statements, not expressions:

```
-- this works
a += 1

-- this doesn't work
print(a += 1)
```

Compound assignments only support a single value on the left and right hand side; additionally, the function calls on the left hand side are only evaluated once:

```
-- calls foo() twice
a[foo()] = a[foo()] + 1

-- calls foo() once
a[foo()] += 1
```

Compound assignments call the arithmetic metamethods (`__add` et al) and table indexing metamethods (`__index` and `__newindex`) as needed - for custom types no extra effort is necessary to support them.

Type annotations

To support gradual typing, Luau supports optional type annotations for variables and functions, as well as declaring type aliases.

Types can be declared for local variables, function arguments and function return types using `:` as a separator:

```
function foo(x: number, y: string): boolean
    local k: string = y:rep(x)
    return k == "a"
end
```

In addition, the type of any expression can be overridden using a type cast `::`:

```
local k = (y :: string):rep(x)
```

There are several simple builtin types: `any` (represents inability of the type checker to reason about the type), `nil`, `boolean`, `number`, `string` and `thread`.

Function types are specified using the arguments and return types, separated with `->`:

```
local foo: (number, string) -> boolean
```

To return no values or more than one, you need to wrap the return type position with parentheses, and then list your types there.

```
local no_returns: (number, string) -> ()
local returns_boolean_and_string: (number, string) -> (boolean, string)

function foo(x: number, y: number): (number, string)
    return x + y, tostring(x) .. tostring(y)
end
```

Note that function types are specified without the argument names in the examples above, but it's also possible to specify the names (that are not semantically significant but can show up in documentation and autocomplete):

```
local callback: (errorCode: number, errorText: string) -> ()
```

Table types are specified using the table literal syntax, using `:` to separate keys from values:

```
local array: { [number] : string }
local object: { x: number, y: string }
```

When the table consists of values keyed by numbers, it's called an array-like table and has a special short-hand syntax, `{T}` (e.g. `{string}`).

Additionally, the type syntax supports type intersections (`((number) -> string) & ((boolean) -> string)`) and unions (`(number | boolean) -> string`). An intersection represents a type with values that conform to both sides at the same time, which is useful for overloaded functions; a union represents a type that can store values of either type - `any` is technically a union of all possible types.

It's common in Lua for function arguments or other values to store either a value of a given type or `nil`; this is represented as a union (`number | nil`), but can be specified using `?` as a shorthand syntax (`number?`).

In addition to declaring types for a given value, Luau supports declaring type aliases via `type` syntax:

```
type Point = { x: number, y: number }
type Array<T> = { [number]: T }
type Something = typeof(string.gmatch("", "%d"))
```

The right hand side of the type alias can be a type definition or a `typeof` expression; `typeof` expression doesn't evaluate its argument at runtime.

By default type aliases are local to the file they are declared in. To be able to use type aliases in other modules using `require`, they need to be exported:

```
export type Point = { x: number, y: number }
```

An exported type can be used in another module by prefixing its name with the require alias that you used to import the module.

```
local M = require(Other.Module)

local a: M.Point = {x=5, y=6}
```

For more information please refer to typechecking documentation (typecheck).

If-then-else expressions

In addition to supporting standard if *statements*, Luau adds support for if *expressions*. Syntactically, `if-then-else` expressions look very similar to if statements. However instead of conditionally executing blocks of code, if expressions conditionally evaluate expressions and return the value produced as a result. Also, unlike if statements, if expressions do not terminate with the `end` keyword.

Here is a simple example of an `if-then-else` expression:

```
local maxValue = if a > b then a else b
```

`if-then-else` expressions may occur in any place a regular expression is used. The `if-then-else` expression must match `if <expr> then <expr> else <expr>`; it can also contain an arbitrary number of `elseif` clauses, like `if <expr> then <expr> elseif <expr> then <expr> else <expr>`. Note that in either case, `else` is mandatory.

Here's is an example demonstrating `elseif`:

```
local sign = if x < 0 then -1 elseif x > 0 then 1 else 0
```

Note: In Luau, the `if-then-else` expression is preferred vs the standard Lua idiom of writing `a and b or c` (which roughly simulates a ternary operator). However, the Lua idiom may return an unexpected result if `b` evaluates to false. The `if-then-else` expression will behave as expected in all situations.

Generalized iteration

Luau uses the standard Lua syntax for iterating through containers, `for vars in values`, but extends the semantics with support for generalized iteration. In Lua, to iterate over a table you need to use an iterator like `next` or a function that returns one like `pairs` or `ipairs`. In Luau, you can simply iterate over a table:

```
for k, v in {1, 4, 9} do
    assert(k * k == v)
end
```

This works for tables but can also be extended for tables or userdata by implementing `__iter` metamethod that is called before the iteration begins, and should return an iterator function like `next` (or a custom one):

```
local obj = { items = {1, 4, 9} }
setmetatable(obj, { __iter = function(o) return next, o.items end })

for k, v in obj do
    assert(k * k == v)
end
```

The default iteration order for tables is specified to be consecutive for elements `1..#t` and unordered after that, visiting every element; similarly to iteration using `pairs`, modifying the table entries for keys other than the current one results in unspecified behavior.

String interpolation

Luau adds an additional way to define string values that allows you to place runtime expressions directly inside specific spots of the literal.

This is a more ergonomic alternative over using `string.format` or `("literal"):format`.

To use string interpolation, use a backtick string literal:

```
local count = 3
print(`Bob has {count} apple(s)!`)
--> Bob has 3 apple(s)!
```

Any expression can be used inside `{}`:

```
local combos = {2, 7, 1, 8, 5}
print(`The lock combination is {table.concat(combos)}.`)
--> The lock combination is 27185.
```

Inside backtick string literal, `\` is used to escape `` ` ``, `{`, `\` itself and a newline:

```
print(`Some example escaping the braces \{like so}`)
--> Some example escaping the braces {like so}

print(`Backslash \ that escapes the space is not a part of the string...`)
--> Backslash  that escapes the space is not a part of the string...

print(`Backslash \\ will escape the second backslash...`)
--> Backslash \ will escape the second backslash...

print(`Some text that also includes \`...`)
--> Some text that also includes `...

local name = "Luau"

print(`Welcome to {
    name
}!`)
--> Welcome to Luau!
```

Restrictions and limitations

The sequence of two opening braces {{"`{{`"}} is rejected with a parse error. This restriction is made to prevent developers using other programming languages with a similar feature from trying to attempt that as a way to escape a single `{` and getting unexpected results in Luau.

Luau currently does not support backtick string literals as a type annotation, so ``type Foo = `Foo` `` is invalid.

Function calls with a backtick string literal without parenthesis is not supported, so ``print`hello` `` is invalid.

Floor division (`//`)

Luau implements support for floor division operator (`//`) for numbers as well as support for `__idiv` metamethod. The syntax and semantics follow Lua 5.3 (https://www.lua.org/manual/5.3/manual.html#3.4.1).

For numbers, `a // b` is equal to `math.floor(a / b)`; when `b` is 0, `a // b` results in infinity or NaN as appropriate.

---

permalink: /typecheck title: Type checking toc: true

Luau supports a gradual type system through the use of type annotations and type inference.

Type inference modes

There are three modes currently available. They must be annotated on the top few lines among the comments.

- `--!nocheck`,
- `--!nonstrict` (default), and
- `--!strict`

`nocheck` mode will simply not start the type inference engine whatsoever.

As for the other two, they are largely similar but with one important difference: in nonstrict mode, we infer `any` for most of the types if we couldn't figure it out early enough. This means that given this snippet:

```
local foo = 1
```

We can infer `foo` to be of type `number`, whereas the `foo` in the snippet below is inferred `any`:

```
local foo
foo = 1
```

However, given the second snippet in strict mode, the type checker would be able to infer `number` for `foo`.

Structural type system

Luau's type system is structural by default, which is to say that we inspect the shape of two tables to see if they are similar enough. This was the obvious choice because Lua 5.1 is inherently structural.

```
type A = {x: number, y: number, z: number?}
type B = {x: number, y: number, z: number}

local a1: A = {x = 1, y = 2}        -- ok
local b1: B = {x = 1, y = 2, z = 3} -- ok

local a2: A = b1 -- ok
local b2: B = a1 -- not ok
```

Builtin types

Lua VM supports 8 primitive types: `nil`, `string`, `number`, `boolean`, `table`, `function`, `thread`, and `userdata`. Of these, `table` and `function` are not represented by name, but have their dedicated syntax as covered in this syntax document (syntax), and `userdata` is represented by concrete types; other types can be specified by their name.

The type checker also provides the builtin types `unknown`, `never`, and `any`.

```
local s = "foo"
local n = 1
local b = true
local t = coroutine.running()

local a: any = 1
print(a.x) -- Type checker believes this to be ok, but crashes at runtime.
```

There's a special case where we intentionally avoid inferring `nil`. It's a good thing because it's never useful for a local variable to always be `nil`, thereby permitting you to assign things to it for Luau to infer that instead.

```
local a
local b = nil
```

`unknown` type

`unknown` is also said to be the *top* type, that is it's a union of all types.

```
local a: unknown = "hello world!"
local b: unknown = 5
local c: unknown = function() return 5 end
```

Unlike `any`, `unknown` will not allow itself to be used as a different type!

```
local function unknown(): unknown
    return if math.random() > 0.5 then "hello world!" else 5
end

local a: string = unknown() -- not ok
local b: number = unknown() -- not ok
local c: string | number = unknown() -- not ok
```

In order to turn a variable of type `unknown` into a different type, you must apply type refinements on that variable.

```
local x = unknown()
if typeof(x) == "number" then
    -- x : number
end
```

`never` type

`never` is also said to be the *bottom* type, meaning there doesn't exist a value that inhabits the type `never`. In fact, it is the *dual* of `unknown`. `never` is useful in many scenarios, and one such use case is when type refinements proves it impossible:

```
local x = unknown()
if typeof(x) == "number" and typeof(x) == "string" then
    -- x : never
end
```

`any` type

`any` is just like `unknown`, except that it allows itself to be used as an arbitrary type without further checks or annotations. Essentially, it's an opt-out from the type system entirely.

```
local x: any = 5
local y: string = x -- no type errors here!
```

Function types

Let's start with something simple.

```
local function f(x) return x end

local a: number = f(1)     -- ok
local b: string = f("foo") -- ok
local c: string = f(true)  -- not ok
```

In strict mode, the inferred type of this function `f` is `<A>(A) -> A` (take a look at generics), whereas in nonstrict we infer `(any) -> any`. We know this is true because `f` can take anything and then return that. If we used `x` with another concrete type, then we would end up inferring that.

Similarly, we can infer the types of the parameters with ease. By passing a parameter into *anything* that also has a type, we are saying "this and that has the same type."

```
local function greetingsHelper(name: string)
    return "Hello, " .. name
end

local function greetings(name)
    return greetingsHelper(name)
end

print(greetings("Alexander"))           -- ok
print(greetings({name = "Alexander"})) -- not ok
```

Table types

From the type checker perspective, each table can be in one of three states. They are: `unsealed table`, `sealed table`, and `generic table`. This is intended to represent how the table's type is allowed to change.

Unsealed tables

An unsealed table is a table which supports adding new properties, which updates the tables type. Unsealed tables are created using table literals. This is one way to accumulate knowledge of the shape of this table.

```
local t = {x = 1} -- {x: number}
t.y = 2           -- {x: number, y: number}
t.z = 3           -- {x: number, y: number, z: number}
```

However, if this local were written as `local t: { x: number } = { x = 1 }`, it ends up sealing the table, so the two assignments henceforth will not be ok.

Furthermore, once we exit the scope where this unsealed table was created in, we seal it.

```
local function vec2(x, y)
    local t = {}
    t.x = x
    t.y = y
    return t
end

local v2 = vec2(1, 2)
v2.z = 3 -- not ok
```

Unsealed tables are *exact* in that any property of the table must be named by the type. Since Luau treats missing properties as having value `nil`, this means that we can treat an unsealed table which does not mention a property as if it mentioned the property, as long as that property is optional.

```
local t = {x = 1}
local u : { x : number, y : number? } = t -- ok because y is optional
local v : { x : number, z : number } = t  -- not ok because z is not optional
```

Sealed tables

A sealed table is a table that is now locked down. This occurs when the table type is spelled out explicitly via a type annotation, or if it is returned from a function.

```
local t : { x: number } = {x = 1}
t.y = 2 -- not ok
```

Sealed tables are *inexact* in that the table may have properties which are not mentioned in the type. As a result, sealed tables support *width subtyping*, which allows a table with more properties to be used as a table with fewer

```
type Point1D = { x : number }
type Point2D = { x : number, y : number }
local p : Point2D = { x = 5, y = 37 }
local q : Point1D = p -- ok because Point2D has more properties than Point1D
```

Generic tables

This typically occurs when the symbol does not have any annotated types or were not inferred anything concrete. In this case, when you index on a parameter, you're requesting that there is a table with a matching interface.

```
local function f(t)
    return t.x + t.y
           --^   --^ {x: _, y: _}
end

f({x = 1, y = 2})        -- ok
f({x = 1, y = 2, z = 3}) -- ok
f({x = 1})               -- not ok
```

Table indexers

These are particularly useful for when your table is used similarly to an array.

```
local t = {"Hello", "world!"} -- {[number]: string}
print(table.concat(t, ", "))
```

Luau supports a concise declaration for array-like tables, `{T}` (for example, `{string}` is equivalent to `{[number]: string}`); the more explicit definition of an indexer is still useful when the key isn't a number, or when the table has other fields like `{ [number]: string, n: number }`.

Generics

The type inference engine was built from the ground up to recognize generics. A generic is simply a type parameter in which another type could be slotted in. It's extremely useful because it allows the type inference engine to remember what the type actually is, unlike `any`.

```
type Pair<T> = {first: T, second: T}

local strings: Pair<string> = {first="Hello", second="World"}
local numbers: Pair<number> = {first=1, second=2}
```

Generic functions

As well as generic type aliases like `Pair<T>`, Luau supports generic functions. These are functions that, as well as their regular data parameters, take type parameters. For example, a function which reverses an array is:

```
function reverse(a)
  local result = {}
  for i = #a, 1, -1 do
    table.insert(result, a[i])
  end
  return result
end
```

The type of this function is that it can reverse an array, and return an array of the same type. Luau can infer this type, but if you want to be explicit, you can declare the type parameter `T`, for example:

```
function reverse<T>(a: {T}): {T}
  local result: {T} = {}
  for i = #a, 1, -1 do
    table.insert(result, a[i])
  end
  return result
end
```

When a generic function is called, Luau infers type arguments, for example

```
local x: {number} = reverse({1, 2, 3})
local y: {string} = reverse({"a", "b", "c"})
```

Generic types are used for built-in functions as well as user functions, for example the type of two-argument `table.insert` is:

```
<T>({T}, T) -> ()
```

Union types

A union type represents *one of* the types in this set. If you try to pass a union onto another thing that expects a *more specific* type, it will fail.

For example, what if this `string | number` was passed into something that expects `number`, but the passed in value was actually a `string`?

```
local stringOrNumber: string | number = "foo"

local onlyString: string = stringOrNumber -- not ok
local onlyNumber: number = stringOrNumber -- not ok
```

Note: it's impossible to be able to call a function if there are two or more function types in this union.

Intersection types

An intersection type represents *all of* the types in this set. It's useful for two main things: to join multiple tables together, or to specify overloadable functions.

```
type XCoord = {x: number}
type YCoord = {y: number}
type ZCoord = {z: number}

type Vector2 = XCoord & YCoord
type Vector3 = XCoord & YCoord & ZCoord

local vec2: Vector2 = {x = 1, y = 2}        -- ok
local vec3: Vector3 = {x = 1, y = 2, z = 3} -- ok
```

```
type SimpleOverloadedFunction = ((string) -> number) & ((number) -> string)

local f: SimpleOverloadedFunction

local r1: number = f("foo") -- ok
local r2: number = f(12345) -- not ok
local r3: string = f("foo") -- not ok
local r4: string = f(12345) -- ok
```

Note: it's impossible to create an intersection type of some primitive types, e.g. `string & number`, or `string & boolean`, or other variations thereof.

Note: Luau still does not support user-defined overloaded functions. Some of Roblox and Lua 5.1 functions have different function signature, so inherently requires overloaded functions.

Singleton types (aka literal types)

Luau's type system also supports singleton types, which means it's a type that represents one single value at runtime. At this time, both string and booleans are representable in types.

> *We do not currently support numbers as types. For now, this is intentional.*

```
local foo: "Foo" = "Foo" -- ok
local bar: "Bar" = foo   -- not ok
local baz: string = foo  -- ok

local t: true = true -- ok
local f: false = false -- ok
```

This happens all the time, especially through type refinements and is also incredibly useful when you want to enforce program invariants in the type system! See tagged unions for more information.

Variadic types

Luau permits assigning a type to the `...` variadic symbol like any other parameter:

```
local function f(...: number)
end

f(1, 2, 3)     -- ok
f(1, "string") -- not ok
```

`f` accepts any number of `number` values.

In type annotations, this is written as `...T`:

```
type F = (...number) -> ...string
```

Type packs

Multiple function return values as well as the function variadic parameter use a type pack to represent a list of types.

When a type alias is defined, generic type pack parameters can be used after the type parameters:

```
type Signal<T, U...> = { f: (T, U...) -> (), data: T }
```

> *Keep in mind that `...T` is a variadic type pack (many elements of the same type `T`), while `U...` is a generic type pack that can contain zero or more types and they don't have to be the same.*

It is also possible for a generic function to reference a generic type pack from the generics list:

```
local function call<T, U...>(s: Signal<T, U...>, ...: U...)
    s.f(s.data, ...)
end
```

Generic types with type packs can be instantiated by providing a type pack:

```
local signal: Signal<string, (number, number, boolean)> = --

call(signal, 1, 2, false)
```

There are also other ways to instantiate types with generic type pack parameters:

```
type A<T, U...> = (T) -> U...

type B = A<number, ...string> -- with a variadic type pack
type C<S...> = A<number, S...> -- with a generic type pack
type D = A<number, ()> -- with an empty type pack
```

Trailing type pack argument can also be provided without parentheses by specifying variadic type arguments:

```
type List<Head, Rest...> = (Head, Rest...) -> ()

type B = List<number> -- Rest... is ()
type C = List<number, string, boolean> -- Rest is (string, boolean)

type Returns<T...> = () -> T...

-- When there are no type parameters, the list can be left empty
type D = Returns<> -- T... is ()
```

Type pack parameters are not limited to a single one, as many as required can be specified:

```
type Callback<Args..., Rets...> = { f: (Args...) -> Rets... }

type A = Callback<(number, string), ...number>
```

Adding types for faux object oriented programs

One common pattern we see with existing Lua/Luau code is the following OO code. While Luau is capable of inferring a decent chunk of this code, it cannot pin down on the types of `self` when it spans multiple methods.

```lua
local Account = {}
Account.__index = Account

function Account.new(name, balance)
    local self = {}
    self.name = name
    self.balance = balance

    return setmetatable(self, Account)
end

-- The `self` type is different from the type returned by `Account.new`
function Account:deposit(credit)
    self.balance += credit
end

-- The `self` type is different from the type returned by `Account.new`
function Account:withdraw(debit)
    self.balance -= debit
end

local account = Account.new("Alexander", 500)
```

For example, the type of `Account.new` is `<a, b>(name: a, balance: b) -> { ..., name: a, balance: b, ... }` (snipping out the metatable). For better or worse, this means you are allowed to call `Account.new(5, "hello")` as well as `Account.new({}, {})`. In this case, this is quite unfortunate, so your first attempt may be to add type annotations to the parameters `name` and `balance`.

There's the next problem: the type of `self` is not shared across methods of `Account`, this is because you are allowed to explicitly opt for a different value to pass as `self` by writing `account.deposit(another_account, 50)`. As a result, the type of `Account:deposit` is `<a, b>(self: { balance: a }, credit: b) -> ()`. Consequently, Luau cannot infer the result of the `+` operation from `a` and `b`, so a type error is reported.

We can see there's a lot of problems happening here. This is a case where you will have to guide Luau, but using the power of top-down type inference you only need to do this in *exactly one* place!

```
type AccountImpl = {
    __index: AccountImpl,
    new: (name: string, balance: number) -> Account,
    deposit: (self: Account, credit: number) -> (),
    withdraw: (self: Account, debit: number) -> (),
}

type Account = typeof(setmetatable({} :: { name: string, balance: number }, {} :: AccountImpl))

-- Only these two annotations are necessary
local Account: AccountImpl = {} :: AccountImpl
Account.__index = Account

-- Using the knowledge of `Account`, we can take in information of the `new` type from `AccountImpl`, so:
-- Account.new :: (name: string, balance: number) -> Account
function Account.new(name, balance)
    local self = {}
    self.name = name
    self.balance = balance

    return setmetatable(self, Account)
end

-- Ditto:
-- Account:deposit :: (self: Account, credit: number) -> ()
function Account:deposit(credit)
    self.balance += credit
end

-- Ditto:
-- Account:withdraw :: (self: Account, debit: number) -> ()
function Account:withdraw(debit)
    self.balance -= debit
end

local account = Account.new("Alexander", 500)
```

Tagged unions

Tagged unions are just union types! In particular, they're union types of tables where they have at least *some* common properties but the structure of the tables are different enough. Here's one example:

```
type Ok<T> = { type: "ok", value: T }
type Err<E> = { type: "err", error: E }
type Result<T, E> = Ok<T> | Err<E>
```

This `Result<T, E>` type can be discriminated by using type refinements on the property `type`, like so:

```
if result.type == "ok" then
    -- result is known to be Ok<T>
    -- and attempting to index for error here will fail
    print(result.value)
elseif result.type == "err" then
    -- result is known to be Err<E>
    -- and attempting to index for value here will fail
    print(result.error)
end
```

Which works out because `value: T` exists only when `type` is in actual fact `"ok"`, and `error: E` exists only when `type` is in actual fact `"err"`.

Type refinements

When we check the type of any lvalue (a global, a local, or a property), what we're doing is we're refining the type, hence "type refinement." The support for this is arbitrarily complex, so go crazy!

Here are all the ways you can refine:

1. Truthy test: `if x then` will refine `x` to be truthy.
2. Type guards: `if type(x) == "number" then` will refine `x` to be `number`.
3. Equality: `x == "hello"` will refine `x` to be a singleton type `"hello"`.

And they can be composed with many of `and`/`or`/`not`. `not`, just like `~=`, will flip the resulting refinements, that is `not x` will refine `x` to be falsy.

Using truthy test:

```
local maybeString: string? = nil

if maybeString then
    local onlyString: string = maybeString -- ok
    local onlyNil: nil = maybeString       -- not ok
end

if not maybeString then
    local onlyString: string = maybeString -- not ok
    local onlyNil: nil = maybeString       -- ok
end
```

Using `type` test:

```
local stringOrNumber: string | number = "foo"

if type(stringOrNumber) == "string" then
    local onlyString: string = stringOrNumber -- ok
    local onlyNumber: number = stringOrNumber -- not ok
end

if type(stringOrNumber) ~= "string" then
    local onlyString: string = stringOrNumber -- not ok
    local onlyNumber: number = stringOrNumber -- ok
end
```

Using equality test:

```
local myString: string = f()

if myString == "hello" then
    local hello: "hello" = myString -- ok because it is absolutely "hello"!
    local copy: string = myString   -- ok
end
```

And as said earlier, we can compose as many of `and`/`or`/`not` as we wish with these refinements:

```
local function f(x: any, y: any)
    if (x == "hello" or x == "bye") and type(y) == "string" then
        -- x is of type "hello" | "bye"
        -- y is of type string
    end

    if not (x ~= "hi") then
        -- x is of type "hi"
    end
end
```

`assert` can also be used to refine in all the same ways:

```
local stringOrNumber: string | number = "foo"

assert(type(stringOrNumber) == "string")

local onlyString: string = stringOrNumber -- ok
local onlyNumber: number = stringOrNumber -- not ok
```

Type casts

Expressions may be typecast using `::`. Typecasting is useful for specifying the type of an expression when the automatically inferred type is too generic.

For example, consider the following table constructor where the intent is to store a table of names:

```
local myTable = {names = {}}
table.insert(myTable.names, 42)         -- Inserting a number ought to cause a type error, but doesn't
```

In order to specify the type of the `names` table a typecast may be used:

```
local myTable = {names = {} :: {string}}
table.insert(myTable.names, 42)         -- not ok, invalid 'number' to 'string' conversion
```

A typecast itself is also type checked to ensure the conversion is made to a subtype of the expression's type or `any`:

```
local numericValue = 1
local value = numericValue :: any             -- ok, all expressions may be cast to 'any'
local flag = numericValue :: boolean          -- not ok, invalid 'number' to 'boolean' conversion
```

Roblox types

Roblox supports a rich set of classes and data types, documented here (https://developer.roblox.com/en-us/api-reference). All of them are readily available for the type checker to use by their name (e.g. `Part` or `RaycastResult`).

When one type inherits from another type, the type checker models this relationship and allows to cast a subclass to the parent class implicitly, so you can pass a `Part` to a function that expects an `Instance`.

All enums are also available to use by their name as part of the `Enum` type library, e.g. `local m: Enum.Material = part.Material`.

Finally, we can automatically deduce what calls like `Instance.new` and `game:GetService` are supposed to return:

```
local part = Instance.new("Part")
local basePart: BasePart = part
```

Note that many of these types provide some properties and methods in both lowerCase and UpperCase; the lowerCase variants are deprecated, and the type system will ask you to use the UpperCase variants instead.

Module interactions

Let's say that we have two modules, `Foo` and `Bar`. Luau will try to resolve the paths if it can find any `require` in any scripts. In this case, when you say `script.Parent.Bar`, Luau will resolve it as: relative to this script, go to my parent and get that script named Bar.

```
-- Module Foo
local Bar = require(script.Parent.Bar)

local baz1: Bar.Baz = 1     -- not ok
local baz2: Bar.Baz = "foo" -- ok

print(Bar.Quux)         -- ok
print(Bar.FakeProperty) -- not ok

Bar.NewProperty = true -- not ok
```

```
-- Module Bar
export type Baz = string

local module = {}

module.Quux = "Hello, world!"

return module
```

There are some caveats here though. For instance, the require path must be resolvable statically, otherwise Luau cannot accurately type check it.

Cyclic module dependencies

Cyclic module dependencies can cause problems for the type checker. In order to break a module dependency cycle a typecast of the module to `any` may be used:

```
local myModule = require(MyModule) :: any
```

---

permalink: /why title: Why Luau?

Around 2006, Roblox (https://www.roblox.com) started using Lua 5.1 as a scripting language for games. Over the years the runtime had to be tweaked to provide a safe, secure sandboxed environment; we gradually started accumulating small library changes and tweaks.

Over the course of the last few years, instead of using Web-based stack for our player-facing application, Lua-based in-game UI and Qt-based editor UI, we've started consolidating a lot of the efforts and developing all of these using Roblox engine and Lua as a scripting language.

Having grown a substantial internal codebase that needed to be correct and performant, and with the focus shifting a bit from novice game developers to professional studios building games on Roblox and our own teams of engineers building applications, there was a need to improve performance and quality of the code we were writing.

Unlike mainline Lua, we also could not afford to do major breaking changes to the language (hence the 5.1 language baseline that remained unchanged for more than a decade). While faster implementations of Lua 5.1 like LuaJIT were available, they didn't meet our needs in terms of portability, ease of change and they didn't address the problem of developing robust code at scale.

All of these motivated us to start reshaping Lua 5.1 that we started from into a new, derivative language that we call Luau. Our focus is on making the language more performant and feature-rich, and make it easier to write robust code through a combination of linting and type checking using a gradual type system.

Complete rewrite?

A very large part of Luau codebase is written from scratch. We needed a set of tools to be able to write language analysis tools; Lua has a parser that is integrated with the bytecode compiler, which makes it unsuitable for complex semantic analysis. For bytecode compilation, while a single pass compiler can deliver better compilation throughput and be simpler than a full frontend/backend, it significantly limits the optimizations that can be done at the bytecode level.

Luau compiler and analysis tools are thus written from scratch, closely following the syntax and semantics of Lua. Our compiler is not single-pass, and instead relies on a set of analysis passes that run over the AST to produce efficient bytecode, followed by some post-process optimizations.

As for the runtime, we had to rewrite the interpreter from scratch to get substantially faster performance; using a combination of techniques pioneered by LuaJIT and custom optimizations that are able to improve performance by taking control over the entire stack (language, compiler, interpreter, virtual machine), we're able to get close to LuaJIT interpreter performance while using C as an implementation language.

The garbage collector and the core libraries represent more of an incremental change, where we used Lua 5.1 as a baseline but we're continuing to rewrite these as well with performance in mind.

While Luau doesn't currently implement JIT/AOT, this is likely to happen at some point; beyond the usual implementation challenges and security concerns, one significant limitation is that we don't have access to JIT on many platforms so for us maintaining excellent interpreted performance for gameplay and application code is more important than reaching peak FLOPS on numerical code.

permalink: /getting-started title: Getting Started toc: true

To get started with Luau you need to use `luau` command line binary to run your code and `luau-analyze` to run static analysis (including type checking and linting). You can download these from a recent release (https://github.com/luau-lang/luau/releases).

Creating a script

To create your own testing script, create a new file with `.luau` as the extension:

```
function ispositive(x)
    return x > 0
end

print(ispositive(1))
print(ispositive("2"))

function isfoo(a)
    return a == "foo"
end

print(isfoo("bar"))
print(isfoo(1))
```

You can now run the file using `luau test.luau` and analyze it using `luau-analyze test.luau`.

Note that there are no warnings about calling `ispositive()` with a string, or calling `isfoo()` a number. This is because the type checking uses non-strict mode by default, which is lenient in how it infers types used by the program.

Type inference

Now modify the script to include `--!strict` at the top:

```
--!strict

function ispositive(x)
    return x > 0
end

print(ispositive(1))
print(ispositive("2"))
```

In `strict` mode, Luau will infer types based on analysis of the code flow. There is also `nonstrict` mode, where analysis is more conservative and types are more frequently inferred as `any` to reduce cases where legitimate code is flagged with warnings.

In this case, Luau will use the `return x > 0` statement to infer that `ispositive()` is a function taking a number and returning a boolean. Note that in this case, it was not necessary to add any explicit type annotations.

Based on Luau's type inference, the analysis tool will now flag the incorrect call to `ispositive()`:

```
$ luau-analyze test.luau
test.luau(7,18): TypeError: Type 'string' could not be converted into 'number'
```

Annotations

You can add annotations to locals, arguments, and function return types. Among other things, annotations can help enforce that you don't accidentally do something stupid. Here's how we would add annotations to `ispositive()`:

```
--!strict

function ispositive(x : number) : boolean
    return x > 0
end

local result : boolean
result = ispositive(1)
```

Now we've told explicitly told Luau that `ispositive()` accepts a number and returns a boolean. This wasn't strictly (pun intended) necessary in this case, because Luau's inference was able to deduce this already. But even in this case, there are advantages to explicit annotations. Imagine that later we decide to change `ispositive()` to return a string value:

```
--!strict

function ispositive(x : number) : boolean
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : boolean
result = ispositive(1)
```

Oops -- we're returning string values, but we forgot to update the function return type. Since we've told Luau that `ispositive()` returns a boolean (and that's how we're using it), the call site isn't flagged as an error. But because the annotation doesn't match our code, we get a warning in the function body itself:

```
$ luau-analyze test.luau
test.luau(5,9): TypeError: Type 'string' could not be converted into 'boolean'
test.luau(7,9): TypeError: Type 'string' could not be converted into 'boolean'
```

The fix is simple; just change the annotation to declare the return type as a string:

```
--!strict

function ispositive(x : number) : string
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : boolean
result = ispositive(1)
```

Well, almost - since we declared `result` as a boolean, the call site is now flagged:

```
$ luau-analyze test.luau
test.luau(12,10): TypeError: Type 'string' could not be converted into 'boolean'
```

If we update the type of the local variable, everything is good. Note that we could also just let Luau infer the type of `result` by changing it to the single line version `local result = ispositive(1)`.

```
--!strict

function ispositive(x : number) : string
    if x > 0 then
        return "yes"
    else
        return "no"
    end
end

local result : string
result = ispositive(1)
```

Conclusions

This has been a brief tour of the basic functionality of Luau, but there's lots more to explore. If you're interested in reading more, check out our main reference pages for syntax (syntax) and typechecking (typecheck).

permalink: /library title: Library toc: true

Luau comes equipped with a standard library of functions designed to manipulate the built-in data types. Note that the library is relatively minimal and doesn't expose ways for scripts to interact with the host environment - it's expected that embedding applications provide extra functionality on top of this and limit or sandbox the system access appropriately, if necessary. For example, Roblox provides a rich API to interact with the 3D environment and limited APIs to interact with external services (https://developer.roblox.com/en-us/api-reference).

This page documents the available builtin libraries and functions. All of these are accessible by default by any script, assuming the host environment exposes them (which is usually a safe assumption outside of extremely constrained environments).

Global functions

While most library functions are provided as part of a library like `table`, a few global functions are exposed without extra namespacing.

```
function assert<T>(value: T, message: string?): T
```

`assert` checks if the value is truthy; if it's not (which means it's `false` or `nil`), it raises an error. The error message can be customized with an optional parameter. Upon success the function returns the `value` argument.

```
function error(obj: any, level: number?)
```

`error` raises an error with the specified object. Note that errors don't have to be strings, although they often are by convention; various error handling mechanisms like `pcall` preserve the error type. When `level` is specified, the error raised is turned into a string that contains call frame information for the caller at level `level`, where `1` refers to the function that called `error`. This can be useful to attribute the errors to callers, for example `error("Expected a valid object", 2)` highlights the caller of the function that called `error` instead of the function itself in the callstack.

```
function gcinfo(): number
```

`gcinfo` returns the total heap size in kilobytes, which includes bytecode objects, global tables as well as the script-allocated objects. Note that Luau uses an incremental garbage collector, and as such at any given point in time the heap may contain both reachable and unreachable objects. The number returned by `gcinfo` reflects the current heap consumption from the operating system perspective and can fluctuate over time as garbage collector frees objects.

```
function getfenv(target: (function | number)?): table
```

Returns the environment table for target function; when `target` is not a function, it must be a number corresponding to the caller stack index, where 1 means the function that calls `getfenv`, and the environment table is returned for the corresponding function from the call stack. When `target` is omitted it defaults to `1`, so `getfenv()` returns the environment table for the calling function.

```
function getmetatable(obj: any): table?
```

Returns the metatable for the specified object; when object is not a table or a userdata, the returned metatable is shared between all objects of the same type. Note that when metatable is protected (has a `__metatable` key), the value corresponding to that key is returned instead and may not be a table.

```
function next<K, V>(t: { [K]: V }, i: K?): (K, V)?
```

Given the table `t`, returns the next key-value pair after `i` in the table traversal order, or nothing if `i` is the last key. When `i` is `nil`, returns the first key-value pair instead.

```
function newproxy(mt: boolean?): userdata
```

Creates a new untyped userdata object; when `mt` is true, the new object has an empty metatable that can be modified using `getmetatable`.

```
function print(args: ...any)
```

Prints all arguments to the standard output, using Tab as a separator.

```
function rawequal(a: any, b: any): boolean
```

Returns true iff `a` and `b` have the same type and point to the same object (for garbage collected types) or are equal (for value types).

```
function rawget<K, V>(t: { [K]: V }, k: K): V?
```

Performs a table lookup with index `k` and returns the resulting value, if present in the table, or nil. This operation bypasses metatables/`__index`.

```
function rawset<K, V>(t: { [K] : V }, k: K, v: V)
```

Assigns table field `k` to the value `v`. This operation bypasses metatables/`__newindex`.

```
function select<T>(i: string, args: ...T): number
function select<T>(i: number, args: ...T): ...T
```

When called with `'#'` as the first argument, returns the number of remaining parameters passed. Otherwise, returns the subset of parameters starting with the specified index. Index can be specified from the start of the arguments (using 1 as the first argument), or from the end (using -1 as the last argument).

```
function setfenv(target: function | number, env: table)
```

Changes the environment table for target function to `env`; when `target` is not a function, it must be a number corresponding to the caller stack index, where 1 means the function that calls `setfenv`, and the environment table is returned for the corresponding function from the call stack.

```
function setmetatable(t: table, mt: table?)
```

Changes metatable for the given table. Note that unlike `getmetatable`, this function only works on tables. If the table already has a protected metatable (has a `__metatable` field), this function errors.

```
function tonumber(s: string, base: number?): number?
```

Converts the input string to the number in base `base` (default 10) and returns the resulting number. If the conversion fails (that is, if the input string doesn't represent a valid number in the specified base), returns `nil` instead.

```
function tostring(obj: any): string
```

Converts the input object to string and returns the result. If the object has a metatable with `__tostring` field, that method is called to perform the conversion.

```
function type(obj: any): string
```

Returns the type of the object, which is one of `"nil"`, `"boolean"`, `"number"`, `"vector"`, `"string"`, `"table"`, `"function"`, `"userdata"` or `"thread"`.

```
function typeof(obj: any): string
```

Returns the type of the object; for userdata objects that have a metatable with the `__type` field *and* are defined by the host (not `newproxy`), returns the value for that key. For custom userdata objects, such as ones returned by `newproxy`, this function returns `"userdata"` to make sure host-defined types can not be spoofed.

```
function ipairs(t: table): <iterator>
```

Returns the triple (generator, state, nil) that can be used to traverse the table using a `for` loop. The traversal results in key-value pairs for the numeric portion of the table; key starts from 1 and increases by 1 on each iteration. The traversal terminates when reaching the first `nil` value (so `ipairs` can't be used to traverse array-like tables with holes).

```
function pairs(t: table): <iterator>
```

Returns the triple (generator, state, nil) that can be used to traverse the table using a `for` loop. The traversal results in key-value pairs for all keys in the table, numeric and otherwise, but doesn't have a defined order.

```
function pcall(f: function, args: ...any): (boolean, ...any)
```

Calls function `f` with parameters `args`. If the function succeeds, returns `true` followed by all return values of `f`. If the function raises an error, returns `false` followed by the error object. Note that `f` can yield, which results in the entire coroutine yielding as well.

```
function xpcall(f: function, e: function, args: ...any): (boolean, ...any)
```

Calls function `f` with parameters `args`. If the function succeeds, returns `true` followed by all return values of `f`. If the function raises an error, calls `e` with the error object as an argument, and returns `false` followed by all return values of `e`. Note that `f` can yield, which results in the entire coroutine yielding as well. `e` can neither yield nor error - if it does raise an error, `xpcall` returns with `false` followed by a special error message.

```
function unpack<V>(a: {V}, f: number?, t: number?): ...V
```

Returns all values of `a` with indices in `[f..t]` range. `f` defaults to 1 and `t` defaults to `#a`. Note that this is equivalent to `table.unpack`.

math library

```
function math.abs(n: number): number
```

Returns the absolute value of `n`. Returns NaN if the input is NaN.

```
function math.acos(n: number): number
```

Returns the arc cosine of `n`, expressed in radians. Returns a value in `[0, pi]` range. Returns NaN if the input is not in `[-1, +1]` range.

```
function math.asin(n: number): number
```

Returns the arc sine of `n`, expressed in radians. Returns a value in `[-pi/2, +pi/2]` range. Returns NaN if the input is not in `[-1, +1]` range.

```
function math.atan2(y: number, x: number): number
```

Returns the arc tangent of `y/x`, expressed in radians. The function takes into account the sign of both arguments in order to determine the quadrant. Returns a value in `[-pi, pi]` range.

```
function math.atan(n: number): number
```

Returns the arc tangent of `n`, expressed in radians. Returns a value in `[-pi/2, pi-2]` range.

```
function math.ceil(n: number): number
```

Rounds `n` upwards to the next integer boundary.

```
function math.cosh(n: number): number
```

Returns the hyperbolic cosine of `n`.

```
function math.cos(n: number): number
```

Returns the cosine of `n`, which is an angle in radians. Returns a value in `[0, 1]` range.

```
function math.deg(n: number): number
```

Converts `n` from radians to degrees and returns the result.

```
function math.exp(n: number): number
```

Returns the base-e exponent of `n`, that is `e^n`.

```
function math.floor(n: number): number
```

Rounds `n` downwards to previous integer boundary.

```
function math.fmod(x: number, y: number): number
```

Returns the remainder of `x` modulo `y`, rounded towards zero. Returns NaN if `y` is zero.

```
function math.frexp(n: number): (number, number)
```

Splits the number into a significand (a number in `[-1, +1]` range) and binary exponent such that `n = s * 2^e`, and returns `s, e`.

```
function math.ldexp(s: number, e: number): number
```

Given the significand and a binary exponent, returns a number `s * 2^e`.

```
function math.log10(n: number): number
```

Returns base-10 logarithm of the input number. Returns NaN if the input is negative, and negative infinity if the input is 0. Equivalent to `math.log(n, 10)`.

```
function math.log(n: number, base: number?): number
```

Returns logarithm of the input number in the specified base; base defaults to `e`. Returns NaN if the input is negative, and negative infinity if the input is 0.

```
function math.max(list: ...number): number
```

Returns the maximum number of the input arguments. The function requires at least one input and will error if zero parameters are passed. If one of the inputs is a NaN, the result may or may not be a NaN.

```
function math.min(list: ...number): number
```

Returns the minimum number of the input arguments. The function requires at least one input and will error if zero parameters are passed. If one of the inputs is a NaN, the result may or may not be a NaN.

```
function math.modf(n: number): (number, number)
```

Returns the integer and fractional part of the input number. Both the integer and fractional part have the same sign as the input number, e.g. `math.modf(-1.5)` returns `-1, -0.5`.

```
function math.pow(x: number, y: number): number
```

Returns `x` raised to the power of `y`.

```
function math.rad(n: number): number
```

Converts `n` from degrees to radians and returns the result.

```
function math.random(): number
function math.random(n: number): number
function math.random(min: number, max: number): number
```

Returns a random number using the global random number generator. A zero-argument version returns a number in `[0, 1]` range. A one-argument version returns a number in `[1, n]` range. A two-argument version returns a number in `[min, max]` range. The input arguments are truncated to integers, so `math.random(1.5)` always returns 1.

```
function math.randomseed(seed: number)
```

Reseeds the global random number generator; subsequent calls to `math.random` will generate a deterministic sequence of numbers that only depends on `seed`.

```
function math.sinh(n: number): number
```

Returns a hyperbolic sine of `n`.

```
function math.sin(n: number): number
```

Returns the sine of `n`, which is an angle in radians. Returns a value in `[0, 1]` range.

```
function math.sqrt(n: number): number
```

Returns the square root of `n`. Returns NaN if the input is negative.

```
function math.tanh(n: number): number
```

Returns the hyperbolic tangent of `n`.

```
function math.tan(n: number): number
```

Returns the tangent of `n`, which is an angle in radians.

```
function math.noise(x: number, y: number?, z: number?): number
```

Returns 3D Perlin noise value for the point `(x, y, z)` (`y` and `z` default to zero if absent). Returns a value in `[-1, 1]` range.

```
function math.clamp(n: number, min: number, max: number): number
```

Returns `n` if the number is in `[min, max]` range; otherwise, returns `min` when `n < min`, and `max` otherwise. If `n` is NaN, may or may not return NaN. The function errors if `min > max`.

```
function math.sign(n: number): number
```

Returns `-1` if `n` is negative, `1` if `n` is positive, and `0` if `n` is zero or NaN.

```
function math.round(n: number): number
```

Rounds `n` to the nearest integer boundary. If `n` is exactly halfway between two integers, rounds `n` away from 0.

table library

```
function table.concat(a: {string}, sep: string?, f: number?, t: number?): string
```

Concatenate all elements of `a` with indices in range `[f..t]` together, using `sep` as a separator if present. `f` defaults to 1 and `t` defaults to `#a`.

```
function table.foreach<K, V, R>(t: { [K]: V }, f: (K, V) -> R?): R?
```

Iterates over all elements of the table in unspecified order; for each key-value pair, calls `f` and returns the result of `f` if it's non-nil. If all invocations of `f` returned `nil`, returns no values. This function has been deprecated and is not recommended for use in new code; use `for` loop instead.

```
function table.foreachi<V, R>(t: {V}, f: (number, V) -> R?): R?
```

Iterates over numeric keys of the table in `[1..#t]` range in order; for each key-value pair, calls `f` and returns the result of `f` if it's non-nil. If all invocations of `f` returned `nil`, returns no values. This function has been deprecated and is not recommended for use in new code; use `for` loop instead.

```
function table.getn<V>(t: {V}): number
```

Returns the length of table `t`. This function has been deprecated and is not recommended for use in new code; use `#t` instead.

```
function table.maxn<V>(t: {V}): number
```

Returns the maximum numeric key of table `t`, or zero if the table doesn't have numeric keys.

```
function table.insert<V>(t: {V}, v: V)
function table.insert<V>(t: {V}, i: number, v: V)
```

When using a two-argument version, appends the value to the array portion of the table (equivalent to `t[#t+1] = v`). When using a three-argument version, inserts the value at index `i` and shifts values at indices after that by 1. `i` should be in `[1..#t]` range.

```
function table.remove<V>(t: {V}, i: number?): V?
```

Removes element `i` from the table and shifts values at indices after that by 1. If `i` is not specified, removes the last element of the table. `i` should be in `[1..#t]` range. Returns the value of the removed element, or `nil` if no element was removed (e.g. table was empty).

```
function table.sort<V>(t: {V}, f: ((V, V) -> boolean)?)
```

Sorts the table `t` in ascending order, using `f` as a comparison predicate: `f` should return `true` iff the first parameter should be before the second parameter in the resulting table. When `f` is not specified, builtin less-than comparison is used instead. The comparison predicate must establish a strict weak ordering - sort results are undefined otherwise.

```
function table.pack<V>(args: ...V): { [number]: V, n: number }
```

Returns a table that consists of all input arguments as array elements, and `n` field that is set to the number of inputs.

```
function table.unpack<V>(a: {V}, f: number?, t: number?): ...V
```

Returns all values of `a` with indices in `[f..t]` range. `f` defaults to 1 and `t` defaults to `#a`.

```
function table.move<V>(a: {V}, f: number, t: number, d: number, tt: {V}?)
```

Copies elements in range `[f..t]` from table `a` to table `tt` if specified and `a` otherwise, starting from the index `d`.

```
function table.create<V>(n: number, v: V?): {V}
```

Creates a table with `n` elements; all of them (range `[1..n]`) are set to `v`. When `v` is nil or omitted, the returned table is empty but has preallocated space for `n` elements which can make subsequent insertions faster. Note that preallocation is only performed for the array portion of the table - using `table.create` on dictionaries is counter-productive.

```
function table.find<V>(t: {V}, v: V): number?
```

Find the first element in the table that is equal to `v` and returns its index; the traversal stops at the first `nil`. If the element is not found, `nil` is returned instead.

```
function table.clear(t: table)
```

Removes all elements from the table while preserving the table capacity, so future assignments don't need to reallocate space.

```
function table.freeze(t: table): table
```

Given a non-frozen table, freezes it such that all subsequent attempts to modify the table or assign its metatable raise an error. If the input table is already frozen or has a protected metatable, the function raises an error; otherwise it returns the input table. Note that the table is frozen in-place and is not being copied. Additionally, only `t` is frozen, and keys/values/metatable of `t` don't change their state and need to be frozen separately if desired.

```
function table.isfrozen(t: table): boolean
```

Returns `true` iff the input table is frozen.

```
function table.clone(t: table): table
```

Returns a copy of the input table that has the same metatable, same keys and values, and is not frozen even if `t` was. The copy is shallow: implementing a deep recursive copy automatically is challenging, and often only certain keys need to be cloned recursively which can be done after the initial clone by modifying the resulting table.

string library

```
function string.byte(s: string, f: number?, t: number?): ...number
```

Returns the numeric code of every byte in the input string with indices in range `[f..t]`. `f` defaults to 1 and `t` defaults to `f`, so a two-argument version of this function returns a single number. If the function is called with a single argument and the argument is out of range, the function returns no values.

```
function string.char(args: ...number): string
```

Returns the string that contains a byte for every input number; all inputs must be integers in `[0..255]` range.

```
function string.find(s: string, p: string, init: number?, plain: boolean?): (number?, number?, ...string)
```

Tries to find an instance of pattern `p` in the string `s`, starting from position `init` (defaults to 1). When `plain` is true, the search is using raw (case-sensitive) string equality, otherwise `p` should be a string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1). If a match is found, returns the position of the match and the length of the match, followed by the pattern captures; otherwise returns `nil`.

```
function string.format(s: string, args: ...any): string
```

Returns a formatted version of the input arguments using a printf-style format string (https://en.cppreference.com/w/c/io/fprintf) `s`. The following format characters are supported:

- `c`: expects an integer number and produces a character with the corresponding character code
- `d`, `i`, `u`: expects an integer number and produces the decimal representation of that number
- `o`: expects an integer number and produces the octal representation of that number
- `x`, `X`: expects an integer number and produces the hexadecimal representation of that number, using lower case or upper case hexadecimal characters
- `e`, `E`, `f`, `g`, `G`: expects a number and produces the floating point representation of that number, using scientific or decimal representation
- `q`: expects a string and produces the same string quoted using double quotation marks, with escaped special characters if necessary
- `s`: expects a string and produces the same string verbatim

The formats support modifiers `-`, `+`, space, `#` and `0`, as well as field width and precision modifiers - with the exception of `*`.

```
function string.gmatch(s: string, p: string): <iterator>
```

Produces an iterator function that, when called repeatedly explicitly or via `for` loop, produces matches of string `s` with string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1) `p`. For every match, the captures within the pattern are returned if present (if a pattern has no captures, the entire matching substring is returned instead).

```
function string.gsub(s: string, p: string, f: function | table | string, maxs: number?): (string, number)
```

For every match of string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1) `p` in `s`, replace the match according to `f`. The substitutions stop after the limit of `maxs`, and the function returns the resulting string followed by the number of substitutions.

When `f` is a string, the substitution uses the string as a replacement. When `f` is a table, the substitution uses the table element with key corresponding to the first pattern capture, if present, and entire match otherwise. Finally, when `f` is a function, the substitution uses the result of calling `f` with call pattern captures, or entire matching substring if no captures are present.

```
function string.len(s: string): number
```

Returns the number of bytes in the string (equivalent to `#s`).

```
function string.lower(s: string): string
```

Returns a string where each byte corresponds to the lower-case ASCII version of the input byte in the source string.

```
function string.match(s: string, p: string, init: number?): ...string?
```

Tries to find an instance of pattern `p` in the string `s`, starting from position `init` (defaults to 1). `p` should be a string pattern (https://www.lua.org/manual/5.3/manual.html#6.4.1). If a match is found, returns all pattern captures, or entire matching substring if no captures are present, otherwise returns `nil`.

```
function string.rep(s: string, n: number): string
```

Returns the input string `s` repeated `n` times. Returns an empty string if `n` is zero or negative.

```
function string.reverse(s: string): string
```

Returns the string with the order of bytes reversed compared to the original. Note that this only works if the input is a binary or ASCII string.

```
function string.sub(s: string, f: number, t: number?): string
```

Returns a substring of the input string with the byte range `[f..t]`; `t` defaults to `#s`, so a two-argument version returns a string suffix.

```
function string.upper(s: string): string
```

Returns a string where each byte corresponds to the upper-case ASCII version of the input byte in the source string.

```
function string.split(s: string, sep: string?): {string}
```

Splits the input string using `sep` as a separator (defaults to `","`) and returns the resulting substrings. If separator is empty, the input string is split into separate one-byte strings.

```
function string.pack(f: string, args: ...any): string
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), encodes all input parameters according to the packing format and returns the resulting string. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

```
function string.packsize(f: string): number
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), returns the size of the resulting packed representation. The pack format can't use variable-length format specifiers. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

```
function string.unpack(f: string, s: string): ...any
```

Given a pack format string (https://www.lua.org/manual/5.3/manual.html#6.4.2), decodes the input string according to the packing format and returns all resulting values. Note that Luau uses fixed sizes for all types that have platform-dependent size in Lua 5.x: short is 16 bit, long is 64 bit, integer is 32-bit and size_t is 32 bit for the purpose of string packing.

coroutine library

```
function coroutine.create(f: function): thread
```

Returns a new coroutine that, when resumed, will run function `f`.

```
function coroutine.running(): thread?
```

Returns the currently running coroutine, or `nil` if the code is running in the main coroutine (depending on the host environment setup, main coroutine may never be used for running code).

```
function coroutine.status(co: thread): string
```

Returns the status of the coroutine, which can be `"running"`, `"suspended"`, `"normal"` or `"dead"`. Dead coroutines have finished their execution and can not be resumed, but their state can still be inspected as they are not dead from the garbage collector point of view.

```
function coroutine.wrap(f: function): function
```

Creates a new coroutine and returns a function that, when called, resumes the coroutine and passes all arguments along to the suspension point. When the coroutine yields or finishes, the wrapped function returns with all values returned at the suspension point.

```
function coroutine.yield(args: ...any): ...any
```

Yields the currently running coroutine and passes all arguments along to the code that resumed the coroutine. The coroutine becomes suspended; when the coroutine is resumed again, the resumption arguments will be forwarded to `yield` which will behave as if it returned all of them.

```
function coroutine.isyieldable(): boolean
```

Returns `true` iff the currently running coroutine can yield. Yielding is prohibited when running inside metamethods like `__index` or C functions like `table.foreach` callback, with the exception of `pcall`/`xpcall`.

```
function coroutine.resume(co: thread, args: ...any): (boolean, ...any)
```

Resumes the coroutine and passes the arguments along to the suspension point. When the coroutine yields or finishes, returns `true` and all values returned at the suspension point. If an error is raised during coroutine resumption, this function returns `false` and the error object, similarly to `pcall`.

```
function coroutine.close(co: thread): (boolean, any?)
```

Closes the coroutine which puts coroutine in the dead state. The coroutine must be dead or suspended - in particular it can't be currently running. If the coroutine that's being closed was in an error state, returns `false` along with an error object; otherwise returns `true`. After closing, the coroutine can't be resumed and the coroutine stack becomes empty.

bit32 library

All functions in the `bit32` library treat input numbers as 32-bit unsigned integers in `[0..4294967295]` range. The bit positions start at 0 where 0 corresponds to the least significant bit.

```
function bit32.arshift(n: number, i: number): number
```

Shifts `n` by `i` bits to the right (if `i` is negative, a left shift is performed instead). The most significant bit of `n` is propagated during the shift. When `i` is larger than 31, returns an integer with all bits set to the sign bit of `n`. When `i` is smaller than `-31`, 0 is returned.

```
function bit32.band(args: ...number): number
```

Performs a bitwise `and` of all input numbers and returns the result. If the function is called with no arguments, an integer with all bits set to 1 is returned.

```
function bit32.bnot(n: number): number
```

Returns a bitwise negation of the input number.

```
function bit32.bor(args: ...number): number
```

Performs a bitwise `or` of all input numbers and returns the result. If the function is called with no arguments, zero is returned.

```
function bit32.bxor(args: ...number): number
```

Performs a bitwise `xor` (exclusive or) of all input numbers and returns the result. If the function is called with no arguments, zero is returned.

```
function bit32.btest(args: ...number): boolean
```

Perform a bitwise `and` of all input numbers, and return `true` iff the result is not 0. If the function is called with no arguments, `true` is returned.

```
function bit32.extract(n: number, f: number, w: number?): number
```

Extracts bits of `n` at position `f` with a width of `w`, and returns the resulting integer. `w` defaults to `1`, so a two-argument version of `extract` returns the bit value at position `f`. Bits are indexed starting at 0. Errors if `f` and `f+w-1` are not between 0 and 31.

```
function bit32.lrotate(n: number, i: number): number
```

Rotates `n` to the left by `i` bits (if `i` is negative, a right rotate is performed instead); the bits that are shifted past the bit width are shifted back from the right.

```
function bit32.lshift(n: number, i: number): number
```

Shifts `n` to the left by `i` bits (if `i` is negative, a right shift is performed instead). When `i` is outside of `[-31..31]` range, returns 0.

```
function bit32.replace(n: number, r: number, f: number, w: number?): number
```

Replaces bits of `n` at position `f` and width `w` with `r`, and returns the resulting integer. `w` defaults to `1`, so a three-argument version of `replace` changes one bit at position `f` to `r` (which should be 0 or 1) and returns the result. Bits are indexed starting at 0. Errors if `f` and `f+w-1` are not between 0 and 31.

```
function bit32.rrotate(n: number, i: number): number
```

Rotates `n` to the right by `i` bits (if `i` is negative, a left rotate is performed instead); the bits that are shifted past the bit width are shifted back from the left.

```
function bit32.rshift(n: number, i: number): number
```

Shifts `n` to the right by `i` bits (if `i` is negative, a left shift is performed instead). When `i` is outside of `[-31..31]` range, returns 0.

```
function bit32.countlz(n: number): number
```

Returns the number of consecutive zero bits in the 32-bit representation of `n` starting from the left-most (most significant) bit. Returns 32 if `n` is zero.

```
function bit32.countrz(n: number): number
```

Returns the number of consecutive zero bits in the 32-bit representation of `n` starting from the right-most (least significant) bit. Returns 32 if `n` is zero.

```
function bit32.byteswap(n: number): number
```

Returns `n` with the order of the bytes swappped.

utf8 library

Strings in Luau can contain arbitrary bytes; however, in many applications strings representing text contain UTF8 encoded data by convention, that can be inspected and manipulated using `utf8` library.

```
function utf8.offset(s: string, n: number, i: number?): number?
```

Returns the byte offset of the Unicode codepoint number `n` in the string, starting from the byte position `i`. When the character is not found, returns `nil` instead.

```
function utf8.codepoint(s: string, i: number?, j: number?): ...number
```

Returns a number for each Unicode codepoint in the string with the starting byte offset in `[i..j]` range. `i` defaults to 1 and `j` defaults to `i`, so a two-argument version of this function returns the Unicode codepoint that starts at byte offset `i`.

```
function utf8.char(args: ...number): string
```

Creates a string by concatenating Unicode codepoints for each input number.

```
function utf8.len(s: string, i: number?, j: number?): number?
```

Returns the number of Unicode codepoints with the starting byte offset in `[i..j]` range, or `nil` followed by the first invalid byte position if the input string is malformed. `i` defaults to 1 and `j` defaults to `#s`, so `utf8.len(s)` returns the number of Unicode codepoints in string `s` or `nil` if the string is malformed.

```
function utf8.codes(s: string): <iterator>
```

Returns an iterator that, when used in `for` loop, produces the byte offset and the codepoint for each Unicode codepoints that `s` consists of.

os library

```
function os.clock(): number
```

Returns a high-precision timestamp (in seconds) that doesn't have a defined baseline, but can be used to measure duration with sub-microsecond precision.

```
function os.date(s: string?, t: number?): table | string
```

Returns the table or string representation of the time specified as `t` (defaults to current time) according to `s` format string.

When `s` starts with `!`, the result uses UTC, otherwise it uses the current timezone.

If `s` is equal to `*t` (or `!*t`), a table representation of the date is returned, with keys `sec`/`min`/`hour` for the time (using 24-hour clock), `day`/`month`/`year` for the date, `wday` for week day (1..7), `yday` for year day (1..366) and `isdst` indicating whether the timezone is currently using daylight savings.

Otherwise, `s` is interpreted as a date format string (https://www.cplusplus.com/reference/ctime/strftime/), with the valid specifiers including any of `aAbBcdHIjmMpSUwWxXyYzZ` or `%`. `s` defaults to `"%c"` so `os.date()` returns the human-readable representation of the current date in local timezone.

```
function os.difftime(a: number, b: number): number
```

Calculates the difference in seconds between `a` and `b`; provided for compatibility only. Please use `a - b` instead.

```
function os.time(t: table?): number
```

When called without arguments, returns the current date/time as a Unix timestamp. When called with an argument, expects it to be a table that contains `sec`/`min`/`hour`/`day`/`month`/`year` keys and returns the Unix timestamp of the specified date/time in UTC.

debug library

```
function debug.info(co: thread, level: number, s: string): ...any
function debug.info(level: number, s: string): ...any
function debug.info(f: function, s: string): ...any
```

Given a stack frame or a function, and a string that specifies the requested information, returns the information about the stack frame or function.

Each character of `s` results in additional values being returned in the same order as the characters appear in the string:

- `s` returns source path for the function
- `l` returns the line number for the stack frame or the line where the function is defined when inspecting a function object
- `n` returns the name of the function, or an empty string if the name is not known
- `f` returns the function object itself
- `a` returns the number of arguments that the function expects followed by a boolean indicating whether the function is variadic or not

For example, `debug.info(2, "sln")` returns source file, current line and function name for the caller of the current function.

```
function debug.traceback(co: thread, msg: string?, level: number?): string
function debug.traceback(msg: string?, level: number?): string
```

Produces a stringified callstack of the given thread, or the current thread, starting with level `level`. If `msg` is specified, then the resulting callstack includes the string before the callstack output, separated with a newline. The format of the callstack is human-readable and subject to change.

buffer library

Buffer is an object that represents a fixed-size mutable block of memory.

All operations on a buffer are provided using the 'buffer' library functions.

Many of the functions accept an offset in bytes from the start of the buffer. Offset of 0 from the start of the buffer memory block accesses the first byte.

All offsets, counts and sizes should be non-negative integer numbers.

If the bytes that are accessed by any read or write operation are outside the buffer memory, an error is thrown.

```
function buffer.create(size: number): buffer
```

Creates a buffer of the requested size with all bytes initialized to 0.

Size limit is 1GB or 1,073,741,824 bytes.

```
function buffer.fromstring(str: string): buffer
```

Creates a buffer initialized to the contents of the string.

The size of the buffer equals to the length of the string.

```
function buffer.tostring(b: buffer): string
```

Returns the buffer data as a string.

```
function buffer.len(b: buffer): number
```

Returns the size of the buffer in bytes.

```
function buffer.readi8(b: buffer, offset: number): number
function buffer.readu8(b: buffer, offset: number): number
function buffer.readi16(b: buffer, offset: number): number
function buffer.readu16(b: buffer, offset: number): number
function buffer.readi32(b: buffer, offset: number): number
function buffer.readu32(b: buffer, offset: number): number
function buffer.readf32(b: buffer, offset: number): number
function buffer.readf64(b: buffer, offset: number): number
```

Used to read the data from the buffer by reinterpreting bytes at the offset as the type in the argument and converting it into a number.

Available types:

Function	Type	Range
readi8	signed 8-bit integer	[-128, 127]
readu8	unsigned 8-bit integer	[0, 255]
readi16	signed 16-bit integer	[-32,768, 32,767]
readu16	unsigned 16-bit integer	[0, 65,535]
readi32	signed 32-bit integer	[-2,147,483,648, 2,147,483,647]

Function	Type	Range
readu32	unsigned 32-bit integer	[0, 4,294,967,295]
readf32	32-bit floating-point number	Single-precision IEEE 754 number
readf64	64-bit floating-point number	Double-precision IEEE 754 number

Floating-point numbers are read and written using a format specified by IEEE 754.

If a floating-point value matches any of bit patterns that represent a NaN (not a number), returned value might be converted to a different quiet NaN representation.

Read and write operations use the little endian byte order.

Integer numbers are read and written using two's complement representation.

```
function buffer.writei8(b: buffer, offset: number, value: number): ()
function buffer.writeu8(b: buffer, offset: number, value: number): ()
function buffer.writei16(b: buffer, offset: number, value: number): ()
function buffer.writeu16(b: buffer, offset: number, value: number): ()
function buffer.writei32(b: buffer, offset: number, value: number): ()
function buffer.writeu32(b: buffer, offset: number, value: number): ()
function buffer.writef32(b: buffer, offset: number, value: number): ()
function buffer.writef64(b: buffer, offset: number, value: number): ()
```

Used to write data to the buffer by converting the number into the type in the argument and reinterpreting it as individual bytes.

Ranges of acceptable values can be seen in the table above.

When writing integers, the number is converted using `bit32` library rules.

Values that are out-of-range will take less significant bits of the full number. For example, writing 43,981 (0xabcd) using writei8 function will take 0xcd and interpret it as an 8-bit signed number -51. It is still recommended to keep all numbers in range of the target type.

Results of converting special number values (inf/nan) to integers are platform-specific.

```
function buffer.readstring(b: buffer, offset: number, count: number): string
```

Used to read a string of length 'count' from the buffer at specified offset.

```
function buffer.writestring(b: buffer, offset: number, value: string, count: number?): ()
```

Used to write data from a string into the buffer at a specified offset.

If an optional 'count' is specified, only 'count' bytes are taken from the string.

Count cannot be larger than the string length.

```
function buffer.copy(target: buffer, targetOffset: number, source: buffer, sourceOffset: number?, count: number?): ()
```

Copy 'count' bytes from 'source' starting at offset 'sourceOffset' into the 'target' at 'targetOffset'.

It is possible for 'source' and 'target' to be the same. Copying an overlapping region inside the same buffer acts as if the source region is copied into a temporary buffer and then that buffer is copied over to the target.

If 'sourceOffset' is nil or is omitted, it defaults to 0.

If 'count' is 'nil' or is omitted, the whole 'source' data starting from 'sourceOffset' is copied.

```
function buffer.fill(b: buffer, offset: number, value: number, count: number?): ()
```

Sets the 'count' bytes in the buffer starting at the specified 'offset' to the 'value'.

If 'count' is 'nil' or is omitted, all bytes from the specified offset until the end of the buffer are set.

---

permalink: /lint title: Linting toc: true

Luau comes with a set of linting passes, that help make sure that the code is correct and consistent. Unlike the type checker, that models the behavior of the code thoroughly and points toward type mismatches that are likely to result in runtime errors, the linter is more opinionated and produces warnings that can often be safely ignored, although it's recommended to keep the code clean of the warnings.

Linter produces many different types of warnings; many of these are enabled by default, and can be suppressed by declaring `--!nolint NAME` at the top of the file. In dire situations `--!nolint` at the top of the file can be used to completely disable all warnings (note that the type checker is still active, and requires a separate `--!nocheck` declaration).

The rest of this page documents all warnings produced by the linter; each warning has a name and a numeric code, the latter is used when displaying warnings.

UnknownGlobal (1)

By default, variables in Luau are global (this is inherited from Lua 5.x and can't be changed because of backwards compatibility). This means that typos in identifiers are invisible to the parser, and often break at runtime. For this reason, the linter considers all globals that aren't part of the builtin global table and aren't explicitly defined in the script "unknown":

```
local displayName = "Roblox"

-- Unknown global 'displaName'
print(displaName)
```

Note that injecting globals via `setfenv` can produce this warning in correct code; global injection is incompatible with type checking and has performance implications so we recommend against it and in favor of using `require` with correctly scoped identifiers.

DeprecatedGlobal (2)

Some global names exist for compatibility but their use is discouraged. This mostly affects globals introduced by Roblox, and since they can have problematic behavior or can break in the future, this warning highlights their uses:

```
-- Global 'ypcall' is deprecated, use 'pcall' instead
ypcall(function()
    print("hello")
end)
```

GlobalUsedAsLocal (3)

The UnknownGlobal lint can catch typos in globals that are read, but can't catch them in globals that are assigned to. Because of this, and to discourage the use of globals in general, linter detects cases when a global is only used in one function and can be safely converted to a local variable. Note that in some cases this requires declaring the local variable in the beginning of the function instead of where it's being assigned to.

```
local function testFunc(a)
    if a < 5 then
        -- Global 'b' is only used in the enclosing function; consider changing it to local
        b = 1
    else
        b = 2
    end
    print(b)
end
```

LocalShadow (4)

In Luau, it is valid to shadow locals and globals with a local variable, including doing it in the same function. This can result in subtle bugs, since the shadowing may not be obvious to the reader. This warning detects cases where local variables shadow other local variables in the same function, or global variables used in the script; for more cases of detected shadowing see `LocalShadowPedantic`.

```
local function foo()
    for i=1,10 do
        -- Variable 'i' shadows previous declaration at line 2
        for i=1,10 do
            print(i)
        end
    end
end
```

SameLineStatement (5)

Luau doesn't require the use of semicolons and doesn't automatically insert them at line breaks. When used wisely this results in code that is easy to read and understand, however it can cause subtle issues and hard to understand code when abused by using many different statements on the same line. This warning highlights cases where code should either be broken into multiple lines, or use `;` as a visual guide.

```
-- A new statement is on the same line; add semi-colon on previous statement to silence
if b < 0 then local a = b + 1 print(a, b) end
```

MultiLineStatement (6)

An opposite problem is having statements that span multiple lines. This is good for readability when the code is indented properly, but when it's not it results in code that's hard to understand, as its easy to confuse the next line for a separate statement.

```
-- Statement spans multiple lines; use indentation to silence
print(math.max(1,
math.min(2, 3)))
```

LocalUnused (7)

This warning is one of the few warnings that highlight unused variables. Local variable declarations that aren't used may indicate a bug in the code (for example, there could be a typo in the code that uses the wrong variable) or redundant code that is no longer necessary (for example, calling a function to get its result and never using this result). This warning warns about locals that aren't used; if the locals are not used intentionally they can be prefixed with _ to silence the warning:

```
local x = 1
-- Variable 'y' is never used; prefix with '_' to silence
local y = 2
print(x, x)
```

FunctionUnused (8)

While unused local variables could be useful for debugging, unused functions usually contain dead code that was meant to be removed. Unused functions clutter code, can be a result of typos similar to local variables, and can mislead the reader into thinking that some functionality is supported.

```
-- Function 'bar' is never used; prefix with '_' to silence
local function bar()
end

local function foo()
end

return foo()
```

ImportUnused (9)

In Luau, there's no first-class module system that's part of the syntax, but `require` function acts as an import statement. When a local is initialized with a `require` result, and the local is unused, this is classified as "unused import". Removing unused imports improves code quality because it makes it obvious what the dependencies of the code are:

```
-- Import 'Roact' is never used; prefix with '_' to silence
local Roact = require(game.Packages.Roact)
```

BuiltinGlobalWrite (10)

While the sandboxing model of Luau prevents overwriting built-in globals such as `table` for the entire program, it's still valid to reassign these globals - this results in "global shadowing", where the script's global table contains a custom version of `table` after writing to it. This is problematic because it disables some optimizations, and can result in misleading code. When shadowing built-in globals, use locals instead.

```
-- Built-in global 'math' is overwritten here; consider using a local or changing the name
math = {}
```

PlaceholderRead (11)

_ variable name is commonly used as a placeholder to discard function results. The linter follows this convention and doesn't warn about the use of _ in various cases where a different name would cause a warning otherwise. To make sure the placeholder is only used to write values to it, this warning detects the cases where it's read instead:

```
local _ = 5
-- Placeholder value '_' is read here; consider using a named variable
return _
```

UnreachableCode (12)

In some cases the linter can detect code that is never executed, because all execution paths through the function exit the function or the loop before reaching it. Such code is misleading because it's not always obvious to the reader that it never runs, and as such it should be removed.

```
function cbrt(v)
    if v >= 0 then
        return v ^ (1/3)
    else
        error('cbrt expects a non-negative argument')
    end
    -- Unreachable code (previous statement always returns)
    return 0
end
```

UnknownType (13)

Luau provides several functions to get the value type as a string (`type`, `typeof`), and some Roblox APIs expose class names through string arguments (`Instance.new`). This warning detects incorrect use of the type names by checking the string literals used in type comparisons and function calls.

```
-- Unknown type 'String' (expected primitive type)
if type(v) == "String" then
    print("v is a string")
end
```

ForRange (14)

When using a numeric for, it's possible to make various mistakes when specifying the for bounds. For example, to iterate through the table backwards, it's important to specify the negative step size. This warning detects several cases where the numeric for only runs for 0 or 1 iterations, or when the step doesn't divide the size of the range evenly.

```
-- For loop should iterate backwards; did you forget to specify -1 as step?
for i=#t,1 do
end
```

UnbalancedAssignment (15)

Assignment statements and local variable declarations in Luau support multiple variables on the left side and multiple values on the right side. The number of values doesn't need to match; when the right side has more values, the extra values are discarded, and then the left side has more variables the extra variables are set to `nil`. However, this can result in subtle bugs where a value is omitted mistakenly. This warning warns about cases like this; if the last expression on the right hand side returns multiple values, the warning is not emitted.

```
-- Assigning 2 values to 3 variables initializes extra variables with nil; add 'nil' to value list to silence
local x, y, z = 1, 2
```

ImplicitReturn (16)

In Luau, there's a subtle difference between returning no values from a function and returning `nil`. In many contexts these are equivalent, but when the results are passed to a variadic function (perhaps implicitly), the difference can be observed - for example, `print(foo())` prints nothing if `foo` returns no values, and `nil` if it returns `nil`.

To help write code that has consistent behavior, linter warns about cases when a function implicitly returns no values, if there are cases when it explicitly returns a result. For code like this it's recommended to use explicit `return` or `return nil` at the end of the function (these have different semantics, so the correct version to use depends on the function):

```
local function find(t, expected)
    for k,v in pairs(t) do
        if k == expected then
            return v
        end
    end
    -- Function 'find' can implicitly return no values even though there's an explicit return at line 4; add explicit return to sile
end
```

DuplicateLocal (17)

Luau syntax allows to use the same name for different parameters of a function as well as different local variables declared in the same statement. This is error prone, even if it's occasionally useful, so a warning is emitted in cases like this, unless the duplicate name is the placeholder _:

```
function foo(a, b, a) -- Function parameter 'a' already defined on column 14
end

function obj:method(self) -- Function parameter 'self' already defined implicitly
end

local x, y, x = v:GetComponents() -- Variable 'x' already defined on column 7
```

FormatString (18)

Luau has several library functions that expect a format string that specifies the behavior for the function. These format strings follow a specific syntax that depends on the question; mistakes in these strings can lead to runtime errors or unexpected behavior of the code.

To help make sure that the strings used for these functions are correct, linter checks calls to `string.format`, `string.pack`, `string.packsize`, `string.unpack`, `string.match`, `string.gmatch`, `string.find`, `string.gsub` and `os.date` and issues warnings when the call uses a literal string with an incorrect format:

```
-- Invalid match pattern: invalid capture reference, must refer to a closed capture
local cap = string.match(s, "(%d)%2")

-- Invalid format string: unfinished format specifier
local str = ("%d %"):format(1, 2)
```

Note that with the exception of `string.format` this only works when the function is called via the library, not via the method call (so prefer `string.match(s, "pattern")` to `s:match("pattern")`).

TableLiteral (19)

Table literals are often used in Luau to create objects or specify data. The syntax for table literals is rich but invites mistakes, for example it allows duplicate keys or redundant index specification for values already present in the list form. This warning is emitted for cases where some entries in the table literal are ignored at runtime as they're duplicate:

```
print({
    first = 1,
    second = 2,
    first = 3, -- Table field 'first' is a duplicate; previously defined at line 2
})
```

UninitializedLocal (20)

It's easy to forget to initialize a local variable and then proceed to use it regardless. This can happen due to a typo, or sometimes it can happen because the original variable definition is shadowed. When a local variable doesn't have an initial value and is used without ever being assigned to, this warning is emitted:

```
local foo

if foo then -- Variable 'foo' defined at line 1 is never initialized or assigned; initialize with 'nil' to silence
        print(foo)
end
```

DuplicateFunction (21)

This warning is emitted when a script defines two functions with the same name in the same scope.

The warning is not produced when the functions are defined in different scopes because this is much more likely to be intentional.

```
function foo() end
function foo() end -- Duplicate function definition: 'foo' also defined on line 1

-- OK: the functions are not defined in the same scope.
if x then
    function bar() end
else
    function bar() end
end
```

DeprecatedApi (22)

This warning is emitted when a script accesses a method or field that is marked as deprecated. Use of deprecated APIs is discouraged since they may have performance or correctness issues, may result in unexpected behavior, and could be removed in the future.

```
function getSize(i: Instance)
    return i.DataCost -- Member 'Instance.DataCost' is deprecated
end
```

TableOperations (23)

To manipulate array-like tables, Luau provides a set of functions as part of the standard `table` library. To help use these functions correctly, this warning is emitted when one of these functions is used incorrectly or can be adjusted to be more performant.

```
local t = {}

table.insert(t, 0, 42) -- table.insert uses index 0 but arrays are 1-based; did you mean 1 instead?
table.insert(t, #t+1, 42) -- table.insert will append the value to the table; consider removing the second argument for efficiency
```

In addition, when type information is present, this warning will be emitted when `#` or `ipairs` is used on a table that has no numeric keys or indexers. This helps avoid common bugs like using `#t == 0` to check if a dictionary is empty.

```
local message = { data = { 1, 2, 3 } }

if #message == 0 then -- Using '#' on a table without an array part is likely a bug
end
```

DuplicateCondition (24)

When checking multiple conditions via `and/or` or `if/elseif`, a copy & paste error may result in checking the same condition redundantly. This almost always indicates a bug, so a warning is emitted when use of a duplicate condition is detected.

```
assert(self._adorns[normID1] and self._adorns[normID1]) -- Condition has already been checked on column 8
```

MisleadingAndOr (25)

In Lua, there is no first-class ternary operator but it can be emulated via `a and b or c` pattern. However, due to how boolean evaluation works, if `b` is `false` or `nil`, the resulting expression evaluates to `c` regardless of the value of `a`. Luau solves this problem with the `if a then b else c` expression; a warning is emitted for and-or expressions where the first alternative is `false` or `nil` because it's almost always a bug.

```
-- The and-or expression always evaluates to the second alternative because the first alternative is false; consider using if-then-el
local x = flag and false or true
```

The code above can be rewritten as follows to avoid the warning and the associated bug:

```
local x = if flag then false else true
```

CommentDirective (26)

Luau uses comments that start from `!` to control certain aspects of analysis, for example setting type checking mode via `--!strict` or disabling individual lints with `--!nolint`. Unknown directives are ignored, for example `--!nostrict` doesn't have any effect on the type checking process as the correct spelling is `--!nonstrict`. This warning flags comment directives that are ignored during processing:

```
--!nostrict
-- Unknown comment directive 'nostrict'; did you mean 'nonstrict'?"
```

IntegerParsing (27)

Luau parses hexadecimal and binary literals as 64-bit integers before converting them to Luau numbers. As a result, numbers that exceed $2^{64 \text{ are silently truncated to } 2}64$, which can result in unexpected program behavior. This warning flags literals that are truncated:

```
-- Hexadecimal number literal exceeded available precision and was truncated to 2^64
local x = 0x1111111111111111111111111111111111111
```

Luau numbers are represented as double precision IEEE754 floating point numbers; they can represent integers up to 2^53 exactly, but larger integers may lose precision. This warning also flags literals that are parsed with a precision loss:

```
-- Number literal exceeded available precision and was truncated to closest representable number
local x = 9007199254740993
```

ComparisonPrecedence (28)

Because of operator precedence rules, not X == Y parses as (not X) == Y; however, often the intent was to invert the result of the comparison. This warning flags erroneous conditions like that, as well as flagging cases where two comparisons happen in a row without any parentheses:

```
-- not X == Y is equivalent to (not X) == Y; consider using X ~= Y, or wrap one of the expressions in parentheses to silence
if not x == 5 then
end

-- X <= Y <= Z is equivalent to (X <= Y) <= Z; wrap one of the expressions in parentheses to silence
if 1 <= x <= 3 then
end
```

title: "Luau News" permalink: /news/ layout: posts sidebar: "none"

permalink: /performance title: Performance toc: true

One of main goals of Luau is to enable high performance code, with gameplay code being the main use case. This can be viewed as two separate goals:

- Make idiomatic code that wasn't tuned faster
- Enable even higher performance through careful tuning

Both of these goals are important - it's insufficient to just focus on the highly tuned code, and all things being equal we prefer to raise all boats by implementing general optimizations. However, in some cases it's important to be aware of optimizations that Luau does and doesn't do.

Worth noting is that Luau is focused on, first and foremost, stable high performance code in interpreted context. This is because JIT compilation is not available on many platforms Luau runs on, and AOT compilation would only work for code that Roblox ships (and even that does not always work). This is in stark contrast with LuaJIT that, while providing an excellent interpreter as well, focuses a lot of the attention on JIT (with many optimizations unavailable in the interpreter).

Luau eventually plans to implement JIT on some platforms, but this is subject to careful memory safety analysis and is likely to not be deployed for client-side scripts, as the extra risk involved in JITs is much more pronounced when it may affect players.

The rest of this document goes into some optimizations that Luau employs and how to best leverage them when writing code. The document is not complete - a lot of optimizations are transparent to the user and involve detailed low-level tuning of various parts that is not described here - and all of this is subject to change without notice, as it doesn't affect the semantics of valid code.

Fast bytecode interpreter

Luau features a very highly tuned portable bytecode interpreter. It's similar to Lua interpreter in that it's written in C, but it's highly tuned to yield efficient assembly when compiled with Clang and latest versions of MSVC. On some workloads it can match the performance of LuaJIT interpreter which is written in highly specialized assembly. We are continuing to tune the interpreter and the bytecode format over time; while extra performance can be extracted by rewriting the interpreter in assembly, we're unlikely to ever do that as the extra gains at this point are marginal, and we gain a lot from C in terms of portability and being able to quickly implement new optimizations.

Of course the interpreter isn't typical C code - it uses many tricks to achieve extreme levels of performance and to coerce the compiler to produce efficient assembly. Due to a better bytecode design and more efficient dispatch loop it's noticeably faster than Lua 5.x (including Lua 5.4 which made some of the changes similar to Luau, but doesn't come close). The bytecode design was partially inspired by excellent LuaJIT interpreter. Most computationally intensive scripts only use the interpreter core loop and builtins, which on x64 compiles into ~16 KB, thus leaving half of the instruction cache for other infrequently called code.

Optimizing compiler

Unlike Lua and LuaJIT, Luau uses a multi-pass compiler with a frontend that parses source into an AST and a backend that generates bytecode from it. This carries a small penalty in terms of compilation time, but results in more flexible code and, crucially, makes it easier to optimize the generated bytecode.

> *Note: Compilation throughput isn't the main focus in Luau, but our compiler is reasonably fast; with all currently implemented optimizations enabled, it compiles 950K lines of Luau code in 1 second on a single core of a desktop Ryzen 5900X CPU, producing bytecode and debug information.*

While bytecode optimizations are limited due to the flexibility of Luau code (e.g. `a * 1` may not be equivalent to `a` if `*` is overloaded through metatables), even in absence of type information Luau compiler can perform some optimizations such as "deep" constant folding across functions and local variables, perform upvalue optimizations for upvalues that aren't mutated, do analysis of builtin function usage, optimize the instruction sequences for multiple variable assignments, and some peephole optimizations on the resulting bytecode. The compiler can also be instructed to use more aggressive optimizations by enabling optimization level 2 (`-O2` in CLI tools), some of which are documented further on this page.

Most bytecode optimizations are performed on individual statements or functions, however the compiler also does a limited amount of interprocedural optimizations; notably, calls to local functions can be optimized with the knowledge of the argument count or number of return values involved. Interprocedural optimizations are limited to a single module due to the compilation model.

Luau compiler currently doesn't use type information to do further optimizations, however early experiments suggest that we can extract further wins. Because we control the entire stack (unlike e.g. TypeScript where the type information is discarded completely before reaching the VM), we have more flexibility there and can make some tradeoffs during codegen even if the type system isn't completely sound. For example, it might be reasonable to assume that in presence of known types, we can infer absence of side effects for arithmetic operations and builtins - if the runtime types mismatch due to intentional violation of the type safety through global injection, the code will still be safely sandboxed; this may unlock optimizations such as common subexpression elimination and allocation hoisting without a JIT. This is speculative pending further research.

Epsilon-overhead debugger

It's important for Luau to have stable and predictable performance. Something that comes up in Lua-based environments often is the use of line hooks to implement debugging (both for breakpoints and for stepping). This is problematic because the support for hooks is typically not free in general, but importantly once the hook is enabled, calling the hook has a considerable overhead, and the hook itself may be very costly to evaluate since it will need to associate the script:line pair with the breakpoint information.

Luau does not support hooks at all, and relies on first-class support for breakpoints (using bytecode patching) and single-stepping (using a custom interpreter loop) to implement debugging. As a result, the presence of breakpoints doesn't slow the script execution down - the only noticeable discrepancy between running code under a debugger and without a debugger should be in cases where breakpoints are evaluated and skipped based on breakpoint conditions, or when stepping over long-running fragments of code.

Inline caching for table and global access

Table access for field lookup is optimized in Luau using a mechanism that blends inline caching (classically used in Java/JavaScript VMs) and HREFs (implemented in LuaJIT). Compiler can predict the hash slot used by field lookup, and the VM can correct this prediction dynamically.

As a result, field access can be very fast in Luau, provided that:

- The field name is known at compile time. To make sure this is the case, `table.field` notation is recommended, although the compiler will also optimize `table["field"]` when the expression is known to be a constant string.
- The field access doesn't use metatables. The fastest way to work with tables in Luau is to store fields directly inside the table, and store methods in the metatable (see below); access to "static" fields in classic OOP designs is best done through `Class.StaticField` instead of `object.StaticField`.
- The object structure is usually uniform. While it's possible to use the same function to access tables of different shape - e.g. `function getX(obj) return obj.x end` can be used on any table that has a field `"x"` - it's best to not vary the keys used in the tables too much, as it defeats this optimization.

The same optimization is applied to the custom globals declared in the script, although it's best to avoid these altogether by using locals instead. Still, this means that the difference between `function` and `local function` is less pronounced in Luau.

Importing global access chains

While global access for library functions can be optimized in a similar way, this optimization breaks down when the global table is using sandboxing through metatables, and even when globals aren't sandboxed, `math.max` still requires two table accesses.

It's always possible to "localize" the global accesses by using `local max = math.max`, but this is cumbersome - in practice it's easy to forget to apply this optimization. To avoid relying on programmers remembering to do this, Luau implements a special optimization called "imports", where most global chains such as `math.max` are resolved when the script is loaded instead of when the script is executed.

This optimization relies on being able to predict the shape of the environment table for a given function; this is possible due to global sandboxing, however this optimization is invalid in some cases:

- `loadstring` can load additional code that runs in context of the caller's environment
- `getfenv`/`setfenv` can directly modify the environment of any function

The use of any of these functions performs a dynamic deoptimization, marking the affected environment as "impure". The optimizations are only in effect on functions with "pure" environments - because of this, the use of `loadstring`/`getfenv`/`setfenv` is not recommended. Note that `getfenv` deoptimizes the environment even if it's only used to read values from the environment.

> *Note: Luau still supports these functions as part of our backwards compatibility promise, although we'd love to switch to Lua 5.2's `_ENV` as that mechanism is cleaner and doesn't require costly dynamic deoptimization.*

Fast method calls

Luau specializes method calls to improve their performance through a combination of compiler, VM and binding optimizations. Compiler emits a specialized instruction sequence when methods are called through `obj:Method` syntax (while this isn't idiomatic anyway, you should avoid `obj.Method(obj)`). When the object in question is a Lua table, VM performs some voodoo magic based on inline caching to try to quickly discover the implementation of this method through the metatable.

For this to be effective, it's crucial that `__index` in a metatable points to a table directly. For performance reasons it's strongly recommended to avoid `__index` functions as well as deep `__index` chains; an ideal object in Luau is a table with a metatable that points to itself through `__index`.

When the object in question is a reflected userdata, a special mechanism called "namecall" is used to minimize the interop cost. In classical Lua binding model, `obj:Method` is called in two steps, retrieving the function object (`obj.Method`) and calling it; both steps are often implemented in C++, and the method retrieval needs to use a method object cache - all of this makes method calls slow.

Luau can directly call the method by name using the "namecall" extension, and an optimized reflection layer can retrieve the correct method quickly through more voodoo magic based on string interning and custom Luau features that aren't exposed through Luau scripts.

As a result of both optimizations, common Lua tricks of caching the method in a local variable aren't very productive in Luau and aren't recommended either.

Specialized builtin function calls

Due to global sandboxing and the ability to dynamically deoptimize code running in impure environments, in pure environments we go beyond optimizing the interpreter and optimize many built-in functions through a "fastcall" mechanism.

For this mechanism to work, function call must be "obvious" to the compiler - it needs to call a builtin function directly, e.g. `math.max(x, 1)`, although it also works if the function is "localized" (`local max = math.max`); this mechanism doesn't work for indirect function calls unless they were inlined during compilation, and doesn't work for method calls (so calling `string.byte` is more efficient than `s:byte`).

The mechanism works by directly invoking a highly specialized and optimized implementation of a builtin function from the interpreter core loop without setting up a stack frame and omitting other work; additionally, some fastcall specializations are partial in that they don't support all types of arguments, for example all `math` library builtins are only specialized for numeric arguments, so calling `math.abs` with a string argument will fall back to the slower implementation that will do string->number coercion.

As a result, builtin calls are very fast in Luau - they are still slightly slower than core instructions such as arithmetic operations, but only slightly so. The set of fastcall builtins is slowly expanding over time and as of this writing contains `assert`, `type`, `typeof`, `rawget`/`rawset`/`rawequal`, `getmetatable`/`setmetatable`, `tonumber`/`tostring`, all functions from `math` (except `noise` and `random`/`randomseed`) and `bit32`, and some functions from `string` and `table` library.

Some builtin functions have partial specializations that reduce the cost of the common case further. Notably:

- `assert` is specialized for cases when the assertion return value is not used and the condition is truthy; this helps reduce the runtime cost of assertions to the extent possible
- `bit32.extract` is optimized further when field and width selectors are constant
- `select` is optimized when the second argument is `...`; in particular, `select(x, ...)` is O(1) when using the builtin dispatch mechanism even though it's normally O(N) in variadic argument count.

Some functions from `math` library like `math.floor` can additionally take advantage of advanced SIMD instruction sets like SSE4.1 when available.

In addition to runtime optimizations for builtin calls, many builtin calls, as well as constants like `math.pi`/`math.huge`, can also be constant-folded by the bytecode compiler when using aggressive optimizations (level 2); this currently applies to most builtin calls with constant arguments and a single return value. For builtin calls that can not be constant folded, compiler assumes knowledge of argument/return count (level 2) to produce more efficient bytecode instructions.

Optimized table iteration

Luau implements a fully generic iteration protocol; however, for iteration through tables in addition to generalized iteration (`for .. in t`) it recognizes three common idioms (`for .. in ipairs(t)`, `for .. in pairs(t)` and `for .. in next, t`) and emits specialized bytecode that is carefully optimized using custom internal iterators.

As a result, iteration through tables typically doesn't result in function calls for every iteration; the performance of iteration using generalized iteration, `pairs` and `ipairs` is comparable, so generalized iteration (without the use of `pairs`/`ipairs`) is recommended unless the code needs to be compatible with vanilla Lua or the specific semantics of `ipairs` (which stops at the first `nil` element) is required. Additionally, using generalized iteration avoids calling `pairs` when the loop starts which can be noticeable when the table is very short.

Iterating through array-like tables using `for i=1,#t` tends to be slightly slower because of extra cost incurred when reading elements from the table.

Optimized table length

Luau tables use a hybrid array/hash storage, like in Lua; in some sense "arrays" don't truly exist and are an internal optimization, but some operations, notably `#t` and functions that depend on it, like `table.insert`, are defined by the Luau/Lua language to allow internal optimizations. Luau takes advantage of that fact.

Unlike Lua, Luau guarantees that the element at index `#t` is stored in the array part of the table. This can accelerate various table operations that use indices limited by `#t`, and this makes `#t` worst-case complexity O(logN), unlike Lua where the worst case complexity is O(N). This also accelerates computation of this value for small tables like `{ [1] = 1 }` since we never need to look at the hash part.

The "default" implementation of `#t` in both Lua and Luau is a binary search. Luau uses a special branch-free (depending on the compiler...) implementation of the binary search which results in 50+% faster computation of table length when it needs to be computed from scratch.

Additionally, Luau can cache the length of the table and adjust it following operations like `table.insert`/`table.remove`; this means that in practice, `#t` is almost always a constant time operation.

Creating and modifying tables

Luau implements several optimizations for table creation. When creating object-like tables, it's recommended to use table literals (`{ ... }`) and to specify all table fields in the literal in one go instead of assigning fields later; this triggers an optimization inspired by LuaJIT's "table templates" and results in higher performance when creating objects. When creating array-like tables, if the maximum size of the table is known up front, it's recommended to use `table.create` function which can create an empty table with preallocated storage, and optionally fill it with a given value.

When the exact table shape isn't known, Luau compiler can still predict the table capacity required in case the table is initialized with an empty literal (`{}`) and filled with fields subsequently. For example, the following code creates a correctly sized table implicitly:

```
local v = {}
v.x = 1
v.y = 2
v.z = 3
return v
```

When appending elements to tables, it's recommended to use `table.insert` (which is the fastest method to append an element to a table if the table size is not known). In cases when a table is filled sequentially, however, it can be more efficient to use a known index for insertion - together with preallocating tables using `table.create` this can result in much faster code, for example this is the fastest way to build a table of squares:

```
local t = table.create(N)


for i=1,N do
        t[i] = i * i
end
```

Native vector math

Luau uses tagged value storage - each value contains a type tag and the data that represents the value of a given type. Because of the need to store 64-bit double precision numbers *and* 64-bit pointers, we don't use NaN tagging and have to pay the cost of 16 bytes per value.

We take advantage of this to provide a native value type that can store a 32-bit floating point vector with 3 components. This type is fundamental to game computations and as such it's important to optimize the storage and the operations with that type - our VM implements first class support for all math operations and component manipulation, which essentially means we have native 3-wide SIMD support. For code that uses many vector values this results in significantly smaller GC pressure and significantly faster execution, and gives programmers a mechanism to hand-vectorize numeric code if need be.

Optimized upvalue storage

Lua implements upvalues as garbage collected objects that can point directly at the thread's stack or, when the value leaves the stack frame (and is "closed"), store the value inside the object. This representation is necessary when upvalues are mutated, but inefficient when they aren't - and 90% or more of upvalues aren't mutated in typical Lua code. Luau takes advantage of this by reworking upvalue storage to prioritize immutable upvalues - capturing upvalues that don't change doesn't require extra allocations or upvalue closing, resulting in faster closure allocation, faster execution, faster garbage collection and faster upvalue access due to better memory locality.

Note that "immutable" in this case only refers to the variable itself - if the variable isn't assigned to it can be captured by value, even if it's a table that has its contents change.

When upvalues are mutable, they do require an extra allocated object; we carefully optimize the memory consumption and access cost for mutable upvalues to reduce the associated overhead.

Closure caching

With optimized upvalue storage, creating new closures (function objects) is more efficient but still requires allocating a new object every time. This can be problematic for cases when functions are passed to algorithms like `table.sort` or functions like `pcall`, as it results in excessive allocation traffic which then leads to more work for garbage collector.

To make closure creation cheaper, Luau compiler implements closure caching - when multiple executions of the same function expression are guaranteed to result in the function object that is semantically identical, the compiler may cache the closure and always return the same object. This changes the function identity which may affect code that uses function objects as table keys, but preserves the calling semantics - compiler will only do this if calling the original (cached) function behaves the same way as a newly created function would. The heuristics used for this optimization are subject to change; currently, the compiler will cache closures that have no upvalues, or all upvalues are immutable (see previous section) and are declared at the module scope, as the module scope is (almost always) evaluated only once.

Fast memory allocator

Similarly to LuaJIT, but unlike vanilla Lua, Luau implements a custom allocator that is highly specialized and tuned to the common allocation workloads we see. The allocator design is inspired by classic pool allocators as well as the excellent `mimalloc`, but through careful domain-specific tuning it beats all general purpose allocators we've tested, including `rpmalloc`, `mimalloc`, `jemalloc`, `ptmalloc` and `tcmalloc`.

This doesn't mean that memory allocation in Luau is free - it's carefully optimized, but it still carries a cost, and a high rate of allocations requires more work from the garbage collector. The garbage collector is incremental, so short of some edge cases this rarely results in visible GC pauses, but can impact the throughput since scripts will interrupt to perform "GC assists" (helping clean up the garbage). Thus for high performance Luau code it's recommended to avoid allocating memory in tight loops, by avoiding temporary table and userdata creation.

In addition to a fast allocator, all frequently used structures in Luau have been optimized for memory consumption, especially on 64-bit platforms, compared to Lua 5.1 baseline. This helps to reduce heap memory footprint and improve performance in some cases by reducing the memory bandwidth impact of garbage collection.

Optimized libraries

While the best performing code in Luau spends most of the time in the interpreter, performance of the standard library functions is critical to some applications. In addition to specializing many small and simple functions using the builtin call mechanism, we spend extra care on optimizing all library functions and providing additional functions beyond the Lua standard library that help achieve good performance with idiomatic code.

Functions from the `table` library like `insert`, `remove` and `move` have been tuned for performance on array-like tables, achieving 3x and more performance compared to un-tuned versions, and Luau provides additional functions like `table.create` and `table.find` to achieve further speedup when applicable. Our implementation of `table.sort` is using `introsort` algorithm which results in guaranteed worst case `NlogN` complexity regardless of the input, and, together with the array-like specializations, helps achieve ~4x speedup on average.

For `string` library, we use a carefully tuned dynamic string buffer implementation; it is optimized for smaller strings to reduce garbage created during string manipulation, and for larger strings it allows to produce a large string without extra copies, especially in cases where the resulting size is known ahead of time. Additionally, functions like `format` have been tuned to avoid the overhead of `sprintf` where possible, resulting in further speedups.

Improved garbage collector pacing

Luau uses an incremental garbage collector which does a little bit of work every so often, and at no point does it stop the world to traverse the entire heap. The runtime will make sure that the collector runs interspersed with the program execution as the program allocates additional memory, which is known as "garbage collection assists", and can also run in response to explicit garbage collection invocation via `lua_gc`. In interactive environments such as video game engines it's possible, and even desirable, to request garbage collection every frame to make sure assists are minimized, since that allows scheduling the garbage collection to run concurrently with other engine processing that doesn't involve script execution.

Inspired by excellent work by Austin Clements on Go's garbage collector pacer, we've implemented a pacing algorithm that uses a proportional–integral–derivative controller to estimate internal garbage collector tunables to reach a target heap size, defined as a percentage of the live heap data (which is more intuitive and actionable than Lua 5.x "GC pause" setting). Luau runtime also estimates the allocation rate making it easy (given uniform allocation rates) to adjust the per-frame garbage collection requests to do most of the required GC work outside of script execution.

Reduced garbage collector pauses

While Luau uses an incremental garbage collector, once per each collector cycle it runs a so-called "atomic" step. While all other GC steps can do very little work by only looking at a few objects at a given time, which means that the collector can have arbitrarily short pauses, the "atomic" step needs to traverse some amount of data that, in some cases, may scale with the application heap. Since atomic step is indivisible, it can result in occasional pauses on the order of tens of milliseconds, which is problematic for interactive applications. We've implemented a series of optimizations to help reduce the atomic step.

Normally objects that have been modified after the GC marked them in an incremental mark phase need to be rescanned during atomic phase, so frequent modifications of existing tables may result in a slow atomic step. To address this, we run a "remark" step where we traverse objects that have been modified after being marked once more (incrementally); additionally, the write barrier that triggers for object modifications changes the transition logic during remark phase to reduce the probability that the object will need to be rescanned.

Another source of scalability challenges is coroutines. Writes to coroutine stacks don't use a write barrier, since that's prohibitively expensive as they are too frequent. This means that coroutine stacks need to be traversed during atomic step, so applications with many coroutines suffer large atomic pauses. To address this, we implement incremental marking of coroutines: marking a coroutine makes it "inactive" and resuming a coroutine (or pushing extra objects on the coroutine stack via C API) makes it "active". Atomic step only needs to traverse active coroutines again, which reduces the cost of atomic step by effectively making coroutine collection incremental as well.

While large tables can be a problem for incremental GC in general since currently marking a single object is indivisible, large weak tables are a unique challenge because they also need to be processed during atomic phase, and the main use case for weak tables - object caches - may result in tables with large capacity but few live objects in long-running applications that exhibit bursts of activity. To address this, weak tables in Luau can be marked as "shrinkable" by including `s` as part of `__mode` string, which results in weak tables being resized to the optimal capacity during GC. This option may result in missing keys during table iteration if the table is resized while iteration is in progress and as such is only recommended for use in specific circumstances.

Optimized garbage collector sweeping

The incremental garbage collector in Luau runs three phases for each cycle: mark, atomic and sweep. Mark incrementally traverses all live objects, atomic finishes various operations that need to happen without mutator intervention (see previous section), and sweep traverses all objects in the heap, reclaiming memory used by dead objects and performing minor fixup for live objects. While objects allocated during the mark phase are traversed in the same cycle and thus may get reclaimed, objects allocated during the sweep phase are considered live. Because of this, the faster the sweep phase completes, the less garbage will accumulate; and, of course, the less time sweeping takes the less overhead there is from this phase of garbage collection on the process.

Since sweeping traverses the whole heap, we maximize the efficiency of this traversal by allocating garbage-collected objects of the same size in 16 KB pages, and traversing each page at a time, which is otherwise known as a paged sweeper. This ensures good locality of reference as consecutively swept objects are contiugous in memory, and allows us to spend no memory for each object on sweep-related data or allocation metadata, since paged sweeper doesn't need to be able to free objects without knowing which page they are in. Compared to linked list based sweeping that Lua/LuaJIT implement, paged sweeper is 2-3x faster, and saves 16 bytes per object on 64-bit platforms.

Function inlining and loop unrolling

By default, the bytecode compiler performs a series of optimizations that result in faster execution of the code, but they preserve both execution semantics and debuggability. For example, a function call is compiled as a function call, which may be observable via `debug.traceback`; a loop is compiled as a loop, which may be observable via `lua_getlocal`. To help improve performance in cases where these restrictions can be relaxed, the bytecode compiler implements additional optimizations when optimization level 2 is enabled (which requires using `-O2` switch when using Luau CLI), namely function inlining and loop unrolling.

Only loops with loop bounds known at compile time, such as `for i=1,4 do`, can be unrolled. The loop body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if unrolling should be performed.

Only local functions (defined either as `local function foo` or `local foo = function`) can be inlined. The function body must be simple enough for the optimization to be profitable; compiler uses heuristics to estimate the performance benefit and automatically decide if each call to the function should be inlined instead. Additionally recursive invocations of a function can’t be inlined at this time, and inlining is completely disabled for modules that use `getfenv`/`setfenv` functions.

In both cases, in addition to removing the overhead associated with function calls or loop iteration, these optimizations can additionally benefit by enabling additional optimizations, such as constant folding of expressions dependent on loop iteration variable or constant function arguments, or using more efficient instructions for certain expressions when the inputs to these instructions are constants.

permalink: /profile title: Profiling toc: true

One of main goals of Luau is to enable high performance code. To help with that goal, we are relentlessly optimizing the compiler and runtime - but ultimately, performance of their code is in developers' hands, and is a combination of good algorithm design and implementation that adheres to the strengths of the language. To help write efficient code, Luau provides a built-in profiler that samples the execution of the program and outputs a profiler dump that can be converted to an interactive flamegraph.

To run the profiler, make sure you have an optimized build of the interpreter (otherwise profiling results are going to be very skewed) and run it with `--profile` argument:

```
$ luau --profile tests/chess.lua
OK      8902    rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
OK      2039    r3k2r/p1ppqpb1/bn2pnp1/3PN3/1p2P3/2N2Q1p/PPPBBPPP/R3K2R w KQkq - 0 0
OK      2812    8/2p5/3p4/KP5r/1R3p1k/8/4P1P1/8 w - - 0 0
OK      9467    r3k2r/Pppp1ppp/1b3nbN/nP6/BBP1P3/q4N2/Pp1P2PP/R2Q1RK1 w kq - 0 1
OK      1486    rnbq1k1r/pp1Pbppp/2p5/8/2B5/8/PPP1NnPP/RNBQK2R w KQ - 1 8
OK      2079    r4rk1/1pp1qppp/p1np1n2/2b1p1B1/2B1P1b1/P1NP1N2/1PP1QPPP/R4RK1 w - - 0 10
Profiler dump written to profile.out (total runtime 2.034 seconds, 20344 samples, 374 stacks)
GC: 0.378 seconds (18.58%), mark 46.80%, remark 3.33%, atomic 1.93%, sweepstring 6.77%, sweep 41.16%
```

The resulting `profile.out` file can be converted to an SVG file by running `perfgraph.py` script that is part of Luau repository:

```
$ python tools/perfgraph.py profile.out >profile.svg
```

This produces an SVG file that can be opened in a browser (the image below is clickable):

profile.svg (/assets/images/chess-profile.svg)

In a flame graph visualization, the individual bars represent function calls, the width represents how much of the total program runtime they execute, and the nesting matches the call stack encountered during program execution. This is a fantastic visualization technique that allows you to hone in on the specific bottlenecks affecting your program performance, optimize those exact bottlenecks, and then re-generate the profile data and visualizer, and look for the next set of true bottlenecks (if any).

Hovering your mouse cursor over individual sections will display detailed function information in the status bar and in a tooltip. If you want to Search for a specific named function, use the Search field in the upper right, or press Ctrl+F.

Notice that some of the bars in the screenshot don't have any text. In some cases, there isn't enough room in the size of the bar to display the name. You can hover your mouse over those bars to see the name and source location of the function in the tool tip, or double-click to zoom in on that part of the flame graph.

Some tooltips will have a source location for the function you're hovering over, but no name. Those are anonymous functions, or functions that were not declared in a way that allows Luau compiler to track the name. To fill in more names, you may want to make these changes to your code:

`local myFunc = function() --[[ work ]] end` -> `local function myFunc() --[[ work ]] end`

Even without these changes, you can hover over a given bar with no visible name and see it's source location.

As any sampling profiler, this profiler relies on gathering enough information for the resulting output to be statistically meaningful. It may miss short functions if they aren't called often enough. By default the profiler runs at 10 kHz, this can be customized by passing a different parameter to `--profile=`. Note that higher frequencies result in higher profiling overhead and longer program execution, potentially skewing the results.

This profiler doesn't track leaf C functions and instead attributes the time spent there to calling Luau functions. As a result, when thinking about why a given function is slow, consider not just the work it does immediately but also the library functions it calls.

This profiler tracks time consumed by Luau thread stacks; when a thread calls another thread via `coroutine.resume`, the time spent is not attributed to the parent thread that's waiting for resume results. This limitation will be removed in the future.

permalink: /sandbox title: Sandboxing toc: true

Luau is safe to embed. Broadly speaking, this means that even in the face of untrusted (and in Roblox case, actively malicious) code, the language and the standard library don't allow unsafe access to the underlying system, and don't have known bugs that allow escaping out of the sandbox (e.g. to gain native code execution through ROP gadgets et al). Additionally, the VM provides extra features to implement isolation of privileged code from unprivileged code and protect one from the other; this is important if the embedding environment decides to expose some APIs that may not be safe to call from untrusted code, for example because they do provide controlled access to the underlying system or risk PII exposure through fingerprinting etc.

This safety is achieved through a combination of removing features from the standard library that are unsafe, adding features to the VM that make it possible to implement sandboxing and isolation, and making sure the implementation is safe from memory safety issues using fuzzing.

Of course, since the entire stack is implemented in C++, the sandboxing isn't formally proven - in theory, compiler or the standard library can have exploitable vulnerabilities. In practice these are very rare and usually found and fixed quickly. While implementing the stack in a safer language such as Rust would make it easier to provide these guarantees, to our knowledge (based on prior art) this would make it difficult to reach the level of performance required.

Library

Parts of the Lua 5.x standard library are unsafe. Some of the functions provide access to the host operating system, including process execution and file reads. Some functions lack sufficient memory safety checks. Some functions are safe if all code is untrusted, but can break the isolation barrier between trusted and untrusted code.

The following libraries and global functions have been removed as a result:

- `io.` library has been removed entirely, as it gives access to files and allows running processes
- `package.` library has been removed entirely, as it gives access to files and allows loading native modules
- `os.` library has been cleaned up from file and environment access functions (`execute`, `exit`, etc.). The only supported functions in the library are `clock`, `date`, `difftime` and `time`.
- `debug.` library has been removed to a large extent, as it has functions that aren't memory safe and other functions break isolation; the only supported functions are `traceback` and `info` (which is similar to `debug.getinfo` but has a slightly different interface).
- `dofile` and `loadfile` allowed access to file system and have been removed.

To achieve memory safety, access to function bytecode has been removed. Bytecode is hard to validate and using untrusted bytecode may lead to exploits. Thus, `loadstring` doesn't work with bytecode inputs, and `string.dump`/`load` have been removed as they aren't necessary anymore. When embedding Luau, bytecode should be encrypted/signed to prevent MITM attacks as well, as the VM assumes that the bytecode was generated by the Luau compiler (which never produces invalid/unsafe bytecode).

Finally, to make isolation possible within the same VM, the following global functions have reduced functionality:

- `collectgarbage` only works with `"count"` argument, as modifying the state of GC can interfere with the expectations of other code running in the process. As such, `collectgarbage()` became an inferior version of `gcinfo()` and is deprecated.
- `newproxy` only works with `true`/`false`/`nil` arguments.
- `module` allowed overriding global packages and was removed as a result.

> *Note: `getfenv`/`setfenv` result in additional isolation challenges, as they allow injecting globals into scripts on the call stack. Ideally, these should be disabled as well, but unfortunately Roblox community relies on these for various reasons. This can be mitigated by limiting interaction between trusted and untrusted code, and/or using separate VMs.*

Environment

The modification to the library functions are sufficient to make embedding safe, but aren't sufficient to provide isolation within the same VM. It should be noted that to achieve guaranteed isolation, it's advisable to load trusted and untrusted code into separate VMs; however, even within the same VM Luau provides additional safety features to make isolation cheaper.

When initializing the default globals table, the tables are protected from modification:

- All libraries (`string`, `math`, etc.) are marked as readonly
- The string metatable is marked as readonly
- The global table itself is marked as readonly

This is using the VM feature that is not accessible from scripts, that prevents all writes to the table, including assignments, `rawset` and `setmetatable`. This makes sure that globals can't be monkey-patched in place, and can only be substituted through `setfenv`.

By itself this would mean that code that runs in Luau can't use globals at all, since assigning globals would fail. While this is feasible, in Roblox we solve this by creating a new global table for each script, that uses `__index` to point to the builtin global table. This safely sandboxes the builtin globals while still allowing writing globals from each script. This also means that short of exposing special shared globals from the host, all scripts are isolated from each other.

`__gc`

Lua 5.1 exposes a `__gc` metamethod for userdata, which can be used on proxies (`newproxy`) to hook into garbage collector. Later versions of Lua extend this mechanism to work on tables.

This mechanism is bad for performance, memory safety and isolation:

- In Lua 5.1, `__gc` support requires traversing userdata lists redundantly during garbage collection to filter out finalizable objects
- In later versions of Lua, userdata that implement `__gc` are split into separate lists; however, finalization prolongs the lifetime of the finalized objects which results in less prompt memory reclamation, and two-step destruction results in extra cache misses for userdata
- `__gc` runs during garbage collection in context of an arbitrary thread which makes the thread identity mechanism used in Roblox to support trusted Luau code invalid
- Objects can be removed from weak tables *after* being finalized, which means that accessing these objects can result in memory safety bugs, unless all exposed userdata methods guard against use-after-gc.
- If `__gc` method ever leaks to scripts, they can call it directly on an object and use any method exposed by that object after that. This means that `__gc` and all other exposed methods must support memory safety when called on a destroyed object.

Because of these issues, Luau does not support `__gc`. Instead it uses tag-based destructors that can perform additional memory cleanup during userdata destruction; crucially, these are only available to the host (so they can never be invoked manually), and they run right before freeing the userdata memory block which is both optimal for performance, and guaranteed to be memory safe.

For monitoring garbage collector behavior the recommendation is to use weak tables instead.

Interrupts

In addition to preventing API access, it can be important for isolation to limit the memory and CPU usage of code that runs inside the VM.

By default, no memory limits are imposed on the running code, so it's possible to exhaust the address space of the host; this is easy to configure from the host for Luau allocations, but of course with a rich API surface exposed by the host it's hard to eliminate this as a possibility. Memory exhaustion doesn't result in memory safety issues or any particular risk to the system that's running the host process, other than the host process getting terminated by the OS.

Limiting CPU usage can be equally challenging with a rich API. However, Luau does provide a VM-level feature to try to contain runaway scripts which makes it possible to terminate any script externally. This works through a global interrupt mechanism, where the host can setup an interrupt handler at any point, and any Luau code is guaranteed to call this handler "eventually" (in practice this can happen at any function call or at any loop iteration). This still leaves the possibility of a very long running script open if the script manages to find a way to call a single C function that takes a lot of time, but short of that the interruption is very prompt.

Roblox sets up the interrupt handler using a watchdog that:

- Limits the runtime of any script in Studio to 10 seconds (configurable through Studio settings)
- Upon client shutdown, interrupts execution of every running script 1 second after shutdown

permalink: /syntax title: Syntax toc: true

Luau uses the baseline syntax of Lua 5.1 (https://www.lua.org/manual/5.1/manual.html#2). For detailed documentation, please refer to the Lua manual, this is an example:

```
local function tree_insert(tree, x)
    local lower, equal, greater = split(tree.root, x)
    if not equal then
        equal = {
            x = x,
            y = math.random(0, 2^31-1),
            left = nil,
            right = nil
        }
    end
    tree.root = merge3(lower, equal, greater)
end
```

Note that future versions of Lua extend the Lua 5.1 syntax with more features; Luau does support string literal extensions but does not support other 5.x additions; for details please refer to compatibility section (compatibility).

The rest of this document documents additional syntax used in Luau.

String literals

Luau implements support for hexadecimal (`\x`), Unicode (`\u`) and `\z` escapes for string literals. This syntax follows Lua 5.3 syntax (https://www.lua.org/manual/5.3/manual.html#3.1):

- `\xAB` inserts a character with the code 0xAB into the string
- `\u{ABC}` inserts a UTF8 byte sequence that encodes U+0ABC character into the string (note that braces are mandatory)
- `\z` at the end of the line inside a string literal ignores all following whitespace including newlines, which can be helpful for breaking long literals into multiple lines.

Number literals

In addition to basic integer and floating-point decimal numbers, Luau supports:

- Hexadecimal integer literals, `0xABC` or `0XABC`
- Binary integer literals, `0b01010101` or `0B01010101`
- Decimal separators in all integer literals, using _ for readability: `1_048_576`, `0xFFFF_FFFF`, `0b_0101_0101`

Note that Luau only has a single number type, a 64-bit IEEE754 double precision number (which can represent integers up to 2^53 exactly), and larger integer literals are stored with precision loss.

Continue statement

In addition to `break` in all loops, Luau supports `continue` statement. Similar to `break`, `continue` must be the last statement in the block.

Note that unlike `break`, `continue` is not a keyword. This is required to preserve backwards compatibility with existing code; so this is a `continue` statement:

```
if x < 0 then
    continue
end
```

Whereas this is a function call:

```
if x < 0 then
    continue()
end
```

When used in `repeat..until` loops, `continue` can not skip the declaration of a local variable if that local variable is used in the loop condition; code like this is invalid and won't compile:

```
repeat
    do continue end
    local a = 5
until a > 0
```

Compound assignments

Luau supports compound assignments with the following operators: `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `..=`. Just like regular assignments, compound assignments are statements, not expressions:

```
-- this works
a += 1

-- this doesn't work
print(a += 1)
```

Compound assignments only support a single value on the left and right hand side; additionally, the function calls on the left hand side are only evaluated once:

```
-- calls foo() twice
a[foo()] = a[foo()] + 1

-- calls foo() once
a[foo()] += 1
```

Compound assignments call the arithmetic metamethods (`__add` et al) and table indexing metamethods (`__index` and `__newindex`) as needed - for custom types no extra effort is necessary to support them.

Type annotations

To support gradual typing, Luau supports optional type annotations for variables and functions, as well as declaring type aliases.

Types can be declared for local variables, function arguments and function return types using `:` as a separator:

```
function foo(x: number, y: string): boolean
    local k: string = y:rep(x)
    return k == "a"
end
```

In addition, the type of any expression can be overridden using a type cast `::`:

```
local k = (y :: string):rep(x)
```

There are several simple builtin types: `any` (represents inability of the type checker to reason about the type), `nil`, `boolean`, `number`, `string` and `thread`.

Function types are specified using the arguments and return types, separated with `->`:

```
local foo: (number, string) -> boolean
```

To return no values or more than one, you need to wrap the return type position with parentheses, and then list your types there.

```
local no_returns: (number, string) -> ()
local returns_boolean_and_string: (number, string) -> (boolean, string)

function foo(x: number, y: number): (number, string)
    return x + y, tostring(x) .. tostring(y)
end
```

Note that function types are specified without the argument names in the examples above, but it's also possible to specify the names (that are not semantically significant but can show up in documentation and autocomplete):

```
local callback: (errorCode: number, errorText: string) -> ()
```

Table types are specified using the table literal syntax, using `:` to separate keys from values:

```
local array: { [number] : string }
local object: { x: number, y: string }
```

When the table consists of values keyed by numbers, it's called an array-like table and has a special short-hand syntax, `{T}` (e.g. `{string}`).

Additionally, the type syntax supports type intersections (`((number) -> string) & ((boolean) -> string)`) and unions (`(number | boolean) -> string`). An intersection represents a type with values that conform to both sides at the same time, which is useful for overloaded functions; a union represents a type that can store values of either type - `any` is technically a union of all possible types.

It's common in Lua for function arguments or other values to store either a value of a given type or `nil`; this is represented as a union (`number | nil`), but can be specified using `?` as a shorthand syntax (`number?`).

In addition to declaring types for a given value, Luau supports declaring type aliases via `type` syntax:

```
type Point = { x: number, y: number }
type Array<T> = { [number]: T }
type Something = typeof(string.gmatch("", "%d"))
```

The right hand side of the type alias can be a type definition or a `typeof` expression; `typeof` expression doesn't evaluate its argument at runtime.

By default type aliases are local to the file they are declared in. To be able to use type aliases in other modules using `require`, they need to be exported:

```
export type Point = { x: number, y: number }
```

An exported type can be used in another module by prefixing its name with the require alias that you used to import the module.

```
local M = require(Other.Module)

local a: M.Point = {x=5, y=6}
```

For more information please refer to typechecking documentation (typecheck).

If-then-else expressions

In addition to supporting standard if *statements*, Luau adds support for if *expressions*. Syntactically, `if-then-else` expressions look very similar to if statements. However instead of conditionally executing blocks of code, if expressions conditionally evaluate expressions and return the value produced as a result. Also, unlike if statements, if expressions do not terminate with the `end` keyword.

Here is a simple example of an `if-then-else` expression:

```
local maxValue = if a > b then a else b
```

`if-then-else` expressions may occur in any place a regular expression is used. The `if-then-else` expression must match `if <expr> then <expr> else <expr>`; it can also contain an arbitrary number of `elseif` clauses, like `if <expr> then <expr> elseif <expr> then <expr> else <expr>`. Note that in either case, `else` is mandatory.

Here's is an example demonstrating `elseif`:

```
local sign = if x < 0 then -1 elseif x > 0 then 1 else 0
```

Note: In Luau, the `if-then-else` expression is preferred vs the standard Lua idiom of writing `a and b or c` (which roughly simulates a ternary operator). However, the Lua idiom may return an unexpected result if `b` evaluates to false. The `if-then-else` expression will behave as expected in all situations.

Generalized iteration

Luau uses the standard Lua syntax for iterating through containers, `for vars in values`, but extends the semantics with support for generalized iteration. In Lua, to iterate over a table you need to use an iterator like `next` or a function that returns one like `pairs` or `ipairs`. In Luau, you can simply iterate over a table:

```
for k, v in {1, 4, 9} do
    assert(k * k == v)
end
```

This works for tables but can also be extended for tables or userdata by implementing `__iter` metamethod that is called before the iteration begins, and should return an iterator function like `next` (or a custom one):

```
local obj = { items = {1, 4, 9} }
setmetatable(obj, { __iter = function(o) return next, o.items end })

for k, v in obj do
    assert(k * k == v)
end
```

The default iteration order for tables is specified to be consecutive for elements `1..#t` and unordered after that, visiting every element; similarly to iteration using `pairs`, modifying the table entries for keys other than the current one results in unspecified behavior.

String interpolation

Luau adds an additional way to define string values that allows you to place runtime expressions directly inside specific spots of the literal.

This is a more ergonomic alternative over using `string.format` or `("literal"):format`.

To use string interpolation, use a backtick string literal:

```
local count = 3
print(`Bob has {count} apple(s)!`)
--> Bob has 3 apple(s)!
```

Any expression can be used inside `{}`:

```
local combos = {2, 7, 1, 8, 5}
print(`The lock combination is {table.concat(combos)}.`)
--> The lock combination is 27185.
```

Inside backtick string literal, `\` is used to escape `` ` ``, `{`, `\` itself and a newline:

```
print(`Some example escaping the braces \{like so}`)
--> Some example escaping the braces {like so}

print(`Backslash \ that escapes the space is not a part of the string...`)
--> Backslash  that escapes the space is not a part of the string...

print(`Backslash \\ will escape the second backslash...`)
--> Backslash \ will escape the second backslash...

print(`Some text that also includes \`...`)
--> Some text that also includes `...

local name = "Luau"

print(`Welcome to {
    name
}!`)
--> Welcome to Luau!
```

Restrictions and limitations

The sequence of two opening braces {{"`{{`"}} is rejected with a parse error. This restriction is made to prevent developers using other programming languages with a similar feature from trying to attempt that as a way to escape a single `{` and getting unexpected results in Luau.

Luau currently does not support backtick string literals as a type annotation, so ``type Foo = `Foo` `` is invalid.

Function calls with a backtick string literal without parenthesis is not supported, so ``print`hello` `` is invalid.

Floor division (`//`)

Luau implements support for floor division operator (`//`) for numbers as well as support for `__idiv` metamethod. The syntax and semantics follow Lua 5.3 (https://www.lua.org/manual/5.3/manual.html#3.4.1).

For numbers, `a // b` is equal to `math.floor(a / b)`; when `b` is 0, `a // b` results in infinity or NaN as appropriate.

---

permalink: /typecheck title: Type checking toc: true

Luau supports a gradual type system through the use of type annotations and type inference.

Type inference modes

There are three modes currently available. They must be annotated on the top few lines among the comments.

- `--!nocheck`,
- `--!nonstrict` (default), and
- `--!strict`

`nocheck` mode will simply not start the type inference engine whatsoever.

As for the other two, they are largely similar but with one important difference: in nonstrict mode, we infer `any` for most of the types if we couldn't figure it out early enough. This means that given this snippet:

```
local foo = 1
```

We can infer `foo` to be of type `number`, whereas the `foo` in the snippet below is inferred `any`:

```
local foo
foo = 1
```

However, given the second snippet in strict mode, the type checker would be able to infer `number` for `foo`.

Structural type system

Luau's type system is structural by default, which is to say that we inspect the shape of two tables to see if they are similar enough. This was the obvious choice because Lua 5.1 is inherently structural.

```
type A = {x: number, y: number, z: number?}
type B = {x: number, y: number, z: number}

local a1: A = {x = 1, y = 2}        -- ok
local b1: B = {x = 1, y = 2, z = 3} -- ok

local a2: A = b1 -- ok
local b2: B = a1 -- not ok
```

Builtin types

Lua VM supports 8 primitive types: `nil`, `string`, `number`, `boolean`, `table`, `function`, `thread`, and `userdata`. Of these, `table` and `function` are not represented by name, but have their dedicated syntax as covered in this syntax document (syntax), and `userdata` is represented by concrete types; other types can be specified by their name.

The type checker also provides the builtin types `unknown`, `never`, and `any`.

```
local s = "foo"
local n = 1
local b = true
local t = coroutine.running()

local a: any = 1
print(a.x) -- Type checker believes this to be ok, but crashes at runtime.
```

There's a special case where we intentionally avoid inferring `nil`. It's a good thing because it's never useful for a local variable to always be `nil`, thereby permitting you to assign things to it for Luau to infer that instead.

```
local a
local b = nil
```

`unknown` type

`unknown` is also said to be the *top* type, that is it's a union of all types.

```
local a: unknown = "hello world!"
local b: unknown = 5
local c: unknown = function() return 5 end
```

Unlike `any`, `unknown` will not allow itself to be used as a different type!

```
local function unknown(): unknown
    return if math.random() > 0.5 then "hello world!" else 5
end

local a: string = unknown() -- not ok
local b: number = unknown() -- not ok
local c: string | number = unknown() -- not ok
```

In order to turn a variable of type `unknown` into a different type, you must apply type refinements on that variable.

```
local x = unknown()
if typeof(x) == "number" then
    -- x : number
end
```

`never` type

`never` is also said to be the *bottom* type, meaning there doesn't exist a value that inhabits the type `never`. In fact, it is the *dual* of `unknown`. `never` is useful in many scenarios, and one such use case is when type refinements proves it impossible:

```
local x = unknown()
if typeof(x) == "number" and typeof(x) == "string" then
    -- x : never
end
```

`any` type

`any` is just like `unknown`, except that it allows itself to be used as an arbitrary type without further checks or annotations. Essentially, it's an opt-out from the type system entirely.

```
local x: any = 5
local y: string = x -- no type errors here!
```

Function types

Let's start with something simple.

```
local function f(x) return x end

local a: number = f(1)     -- ok
local b: string = f("foo") -- ok
local c: string = f(true)  -- not ok
```

In strict mode, the inferred type of this function `f` is `<A>(A) -> A` (take a look at generics), whereas in nonstrict we infer `(any) -> any`. We know this is true because `f` can take anything and then return that. If we used `x` with another concrete type, then we would end up inferring that.

Similarly, we can infer the types of the parameters with ease. By passing a parameter into *anything* that also has a type, we are saying "this and that has the same type."

```
local function greetingsHelper(name: string)
    return "Hello, " .. name
end

local function greetings(name)
    return greetingsHelper(name)
end

print(greetings("Alexander"))          -- ok
print(greetings({name = "Alexander"})) -- not ok
```

Table types

From the type checker perspective, each table can be in one of three states. They are: `unsealed table`, `sealed table`, and `generic table`. This is intended to represent how the table's type is allowed to change.

Unsealed tables

An unsealed table is a table which supports adding new properties, which updates the tables type. Unsealed tables are created using table literals. This is one way to accumulate knowledge of the shape of this table.

```
local t = {x = 1} -- {x: number}
t.y = 2           -- {x: number, y: number}
t.z = 3           -- {x: number, y: number, z: number}
```

However, if this local were written as `local t: { x: number } = { x = 1 }`, it ends up sealing the table, so the two assignments henceforth will not be ok.

Furthermore, once we exit the scope where this unsealed table was created in, we seal it.

```
local function vec2(x, y)
    local t = {}
    t.x = x
    t.y = y
    return t
end

local v2 = vec2(1, 2)
v2.z = 3 -- not ok
```

Unsealed tables are *exact* in that any property of the table must be named by the type. Since Luau treats missing properties as having value `nil`, this means that we can treat an unsealed table which does not mention a property as if it mentioned the property, as long as that property is optional.

```
local t = {x = 1}
local u : { x : number, y : number? } = t -- ok because y is optional
local v : { x : number, z : number } = t  -- not ok because z is not optional
```

Sealed tables

A sealed table is a table that is now locked down. This occurs when the table type is spelled out explicitly via a type annotation, or if it is returned from a function.

```
local t : { x: number } = {x = 1}
t.y = 2 -- not ok
```

Sealed tables are *inexact* in that the table may have properties which are not mentioned in the type. As a result, sealed tables support *width subtyping*, which allows a table with more properties to be used as a table with fewer

```
type Point1D = { x : number }
type Point2D = { x : number, y : number }
local p : Point2D = { x = 5, y = 37 }
local q : Point1D = p -- ok because Point2D has more properties than Point1D
```

Generic tables

This typically occurs when the symbol does not have any annotated types or were not inferred anything concrete. In this case, when you index on a parameter, you're requesting that there is a table with a matching interface.

```
local function f(t)
    return t.x + t.y
           --^   --^ {x: _, y: _}
end

f({x = 1, y = 2})         -- ok
f({x = 1, y = 2, z = 3}) -- ok
f({x = 1})                -- not ok
```

Table indexers

These are particularly useful for when your table is used similarly to an array.

```
local t = {"Hello", "world!"} -- {[number]: string}
print(table.concat(t, ", "))
```

Luau supports a concise declaration for array-like tables, `{T}` (for example, `{string}` is equivalent to `{[number]: string}`); the more explicit definition of an indexer is still useful when the key isn't a number, or when the table has other fields like `{ [number]: string, n: number }`.

Generics

The type inference engine was built from the ground up to recognize generics. A generic is simply a type parameter in which another type could be slotted in. It's extremely useful because it allows the type inference engine to remember what the type actually is, unlike `any`.

```
type Pair<T> = {first: T, second: T}

local strings: Pair<string> = {first="Hello", second="World"}
local numbers: Pair<number> = {first=1, second=2}
```

Generic functions

As well as generic type aliases like `Pair<T>`, Luau supports generic functions. These are functions that, as well as their regular data parameters, take type parameters. For example, a function which reverses an array is:

```
function reverse(a)
  local result = {}
  for i = #a, 1, -1 do
    table.insert(result, a[i])
  end
  return result
end
```

The type of this function is that it can reverse an array, and return an array of the same type. Luau can infer this type, but if you want to be explicit, you can declare the type parameter `T`, for example:

```
function reverse<T>(a: {T}): {T}
  local result: {T} = {}
  for i = #a, 1, -1 do
    table.insert(result, a[i])
  end
  return result
end
```

When a generic function is called, Luau infers type arguments, for example

```
local x: {number} = reverse({1, 2, 3})
local y: {string} = reverse({"a", "b", "c"})
```

Generic types are used for built-in functions as well as user functions, for example the type of two-argument `table.insert` is:

```
<T>({T}, T) -> ()
```

Union types

A union type represents *one of* the types in this set. If you try to pass a union onto another thing that expects a *more specific* type, it will fail.

For example, what if this `string | number` was passed into something that expects `number`, but the passed in value was actually a `string`?

```
local stringOrNumber: string | number = "foo"

local onlyString: string = stringOrNumber -- not ok
local onlyNumber: number = stringOrNumber -- not ok
```

Note: it's impossible to be able to call a function if there are two or more function types in this union.

Intersection types

An intersection type represents *all of* the types in this set. It's useful for two main things: to join multiple tables together, or to specify overloadable functions.

```
type XCoord = {x: number}
type YCoord = {y: number}
type ZCoord = {z: number}

type Vector2 = XCoord & YCoord
type Vector3 = XCoord & YCoord & ZCoord

local vec2: Vector2 = {x = 1, y = 2}        -- ok
local vec3: Vector3 = {x = 1, y = 2, z = 3} -- ok
```

```
type SimpleOverloadedFunction = ((string) -> number) & ((number) -> string)

local f: SimpleOverloadedFunction

local r1: number = f("foo") -- ok
local r2: number = f(12345) -- not ok
local r3: string = f("foo") -- not ok
local r4: string = f(12345) -- ok
```

Note: it's impossible to create an intersection type of some primitive types, e.g. `string & number`, or `string & boolean`, or other variations thereof.

Note: Luau still does not support user-defined overloaded functions. Some of Roblox and Lua 5.1 functions have different function signature, so inherently requires overloaded functions.

Singleton types (aka literal types)

Luau's type system also supports singleton types, which means it's a type that represents one single value at runtime. At this time, both string and booleans are representable in types.

> *We do not currently support numbers as types. For now, this is intentional.*

```
local foo: "Foo" = "Foo" -- ok
local bar: "Bar" = foo   -- not ok
local baz: string = foo  -- ok

local t: true = true -- ok
local f: false = false -- ok
```

This happens all the time, especially through type refinements and is also incredibly useful when you want to enforce program invariants in the type system! See tagged unions for more information.

Variadic types

Luau permits assigning a type to the `...` variadic symbol like any other parameter:

```
local function f(...: number)
end

f(1, 2, 3)      -- ok
f(1, "string") -- not ok
```

`f` accepts any number of `number` values.

In type annotations, this is written as `...T`:

```
type F = (...number) -> ...string
```

Type packs

Multiple function return values as well as the function variadic parameter use a type pack to represent a list of types.

When a type alias is defined, generic type pack parameters can be used after the type parameters:

```
type Signal<T, U...> = { f: (T, U...) -> (), data: T }
```

> *Keep in mind that `...T` is a variadic type pack (many elements of the same type `T`), while `U...` is a generic type pack that can contain zero or more types and they don't have to be the same.*

It is also possible for a generic function to reference a generic type pack from the generics list:

```
local function call<T, U...>(s: Signal<T, U...>, ...: U...)
    s.f(s.data, ...)
end
```

Generic types with type packs can be instantiated by providing a type pack:

```
local signal: Signal<string, (number, number, boolean)> = --

call(signal, 1, 2, false)
```

There are also other ways to instantiate types with generic type pack parameters:

```
type A<T, U...> = (T) -> U...

type B = A<number, ...string> -- with a variadic type pack
type C<S...> = A<number, S...> -- with a generic type pack
type D = A<number, ()> -- with an empty type pack
```

Trailing type pack argument can also be provided without parentheses by specifying variadic type arguments:

```
type List<Head, Rest...> = (Head, Rest...) -> ()

type B = List<number> -- Rest... is ()
type C = List<number, string, boolean> -- Rest is (string, boolean)

type Returns<T...> = () -> T...

-- When there are no type parameters, the list can be left empty
type D = Returns<> -- T... is ()
```

Type pack parameters are not limited to a single one, as many as required can be specified:

```
type Callback<Args..., Rets...> = { f: (Args...) -> Rets... }

type A = Callback<(number, string), ...number>
```

Adding types for faux object oriented programs

One common pattern we see with existing Lua/Luau code is the following OO code. While Luau is capable of inferring a decent chunk of this code, it cannot pin down on the types of `self` when it spans multiple methods.

```
local Account = {}
Account.__index = Account

function Account.new(name, balance)
    local self = {}
    self.name = name
    self.balance = balance

    return setmetatable(self, Account)
end

-- The `self` type is different from the type returned by `Account.new`
function Account:deposit(credit)
    self.balance += credit
end

-- The `self` type is different from the type returned by `Account.new`
function Account:withdraw(debit)
    self.balance -= debit
end

local account = Account.new("Alexander", 500)
```

For example, the type of `Account.new` is `<a, b>(name: a, balance: b) -> { ..., name: a, balance: b, ... }` (snipping out the metatable). For better or worse, this means you are allowed to call `Account.new(5, "hello")` as well as `Account.new({}, {})`. In this case, this is quite unfortunate, so your first attempt may be to add type annotations to the parameters `name` and `balance`.

There's the next problem: the type of `self` is not shared across methods of `Account`, this is because you are allowed to explicitly opt for a different value to pass as `self` by writing `account.deposit(another_account, 50)`. As a result, the type of `Account:deposit` is `<a, b>(self: { balance: a }, credit: b) -> ()`. Consequently, Luau cannot infer the result of the `+` operation from `a` and `b`, so a type error is reported.

We can see there's a lot of problems happening here. This is a case where you will have to guide Luau, but using the power of top-down type inference you only need to do this in *exactly one* place!

```
type AccountImpl = {
    __index: AccountImpl,
    new: (name: string, balance: number) -> Account,
    deposit: (self: Account, credit: number) -> (),
    withdraw: (self: Account, debit: number) -> (),
}

type Account = typeof(setmetatable({} :: { name: string, balance: number }, {} :: AccountImpl))

-- Only these two annotations are necessary
local Account: AccountImpl = {} :: AccountImpl
Account.__index = Account

-- Using the knowledge of `Account`, we can take in information of the `new` type from `AccountImpl`, so:
-- Account.new :: (name: string, balance: number) -> Account
function Account.new(name, balance)
    local self = {}
    self.name = name
    self.balance = balance

    return setmetatable(self, Account)
end

-- Ditto:
-- Account:deposit :: (self: Account, credit: number) -> ()
function Account:deposit(credit)
    self.balance += credit
end

-- Ditto:
-- Account:withdraw :: (self: Account, debit: number) -> ()
function Account:withdraw(debit)
    self.balance -= debit
end

local account = Account.new("Alexander", 500)
```

Tagged unions

Tagged unions are just union types! In particular, they're union types of tables where they have at least *some* common properties but the structure of the tables are different enough. Here's one example:

```
type Ok<T> = { type: "ok", value: T }
type Err<E> = { type: "err", error: E }
type Result<T, E> = Ok<T> | Err<E>
```

This `Result<T, E>` type can be discriminated by using type refinements on the property `type`, like so:

```
if result.type == "ok" then
    -- result is known to be Ok<T>
    -- and attempting to index for error here will fail
    print(result.value)
elseif result.type == "err" then
    -- result is known to be Err<E>
    -- and attempting to index for value here will fail
    print(result.error)
end
```

Which works out because `value: T` exists only when `type` is in actual fact `"ok"`, and `error: E` exists only when `type` is in actual fact `"err"`.

Type refinements

When we check the type of any lvalue (a global, a local, or a property), what we're doing is we're refining the type, hence "type refinement." The support for this is arbitrarily complex, so go crazy!

Here are all the ways you can refine:

1. Truthy test: `if x then` will refine `x` to be truthy.
2. Type guards: `if type(x) == "number" then` will refine `x` to be `number`.
3. Equality: `x == "hello"` will refine `x` to be a singleton type `"hello"`.

And they can be composed with many of `and`/`or`/`not`. `not`, just like `~=`, will flip the resulting refinements, that is `not x` will refine `x` to be falsy.

Using truthy test:

```
local maybeString: string? = nil

if maybeString then
    local onlyString: string = maybeString -- ok
    local onlyNil: nil = maybeString       -- not ok
end

if not maybeString then
    local onlyString: string = maybeString -- not ok
    local onlyNil: nil = maybeString       -- ok
end
```

Using `type` test:

```
local stringOrNumber: string | number = "foo"

if type(stringOrNumber) == "string" then
    local onlyString: string = stringOrNumber -- ok
    local onlyNumber: number = stringOrNumber -- not ok
end

if type(stringOrNumber) ~= "string" then
    local onlyString: string = stringOrNumber -- not ok
    local onlyNumber: number = stringOrNumber -- ok
end
```

Using equality test:

```
local myString: string = f()

if myString == "hello" then
    local hello: "hello" = myString -- ok because it is absolutely "hello"!
    local copy: string = myString   -- ok
end
```

And as said earlier, we can compose as many of `and`/`or`/`not` as we wish with these refinements:

```
local function f(x: any, y: any)
    if (x == "hello" or x == "bye") and type(y) == "string" then
        -- x is of type "hello" | "bye"
        -- y is of type string
    end

    if not (x ~= "hi") then
        -- x is of type "hi"
    end
end
```

`assert` can also be used to refine in all the same ways:

```
local stringOrNumber: string | number = "foo"

assert(type(stringOrNumber) == "string")

local onlyString: string = stringOrNumber -- ok
local onlyNumber: number = stringOrNumber -- not ok
```

Type casts

Expressions may be typecast using `::`. Typecasting is useful for specifying the type of an expression when the automatically inferred type is too generic.

For example, consider the following table constructor where the intent is to store a table of names:

```
local myTable = {names = {}}
table.insert(myTable.names, 42)         -- Inserting a number ought to cause a type error, but doesn't
```

In order to specify the type of the `names` table a typecast may be used:

```
local myTable = {names = {} :: {string}}
table.insert(myTable.names, 42)         -- not ok, invalid 'number' to 'string' conversion
```

A typecast itself is also type checked to ensure the conversion is made to a subtype of the expression's type or `any`:

```
local numericValue = 1
local value = numericValue :: any             -- ok, all expressions may be cast to 'any'
local flag = numericValue :: boolean          -- not ok, invalid 'number' to 'boolean' conversion
```

Roblox types

Roblox supports a rich set of classes and data types, documented here (https://developer.roblox.com/en-us/api-reference). All of them are readily available for the type checker to use by their name (e.g. `Part` or `RaycastResult`).

When one type inherits from another type, the type checker models this relationship and allows to cast a subclass to the parent class implicitly, so you can pass a `Part` to a function that expects an `Instance`.

All enums are also available to use by their name as part of the `Enum` type library, e.g. `local m: Enum.Material = part.Material`.

Finally, we can automatically deduce what calls like `Instance.new` and `game:GetService` are supposed to return:

```
local part = Instance.new("Part")
local basePart: BasePart = part
```

Note that many of these types provide some properties and methods in both lowerCase and UpperCase; the lowerCase variants are deprecated, and the type system will ask you to use the UpperCase variants instead.

Module interactions

Let's say that we have two modules, `Foo` and `Bar`. Luau will try to resolve the paths if it can find any `require` in any scripts. In this case, when you say `script.Parent.Bar`, Luau will resolve it as: relative to this script, go to my parent and get that script named Bar.

```
-- Module Foo
local Bar = require(script.Parent.Bar)

local baz1: Bar.Baz = 1     -- not ok
local baz2: Bar.Baz = "foo" -- ok

print(Bar.Quux)         -- ok
print(Bar.FakeProperty) -- not ok

Bar.NewProperty = true -- not ok
```

```
-- Module Bar
export type Baz = string

local module = {}

module.Quux = "Hello, world!"

return module
```

There are some caveats here though. For instance, the require path must be resolvable statically, otherwise Luau cannot accurately type check it.

Cyclic module dependencies

Cyclic module dependencies can cause problems for the type checker. In order to break a module dependency cycle a typecast of the module to `any` may be used:

```
local myModule = require(MyModule) :: any
```

---

permalink: /why title: Why Luau?

Around 2006, Roblox (https://www.roblox.com) started using Lua 5.1 as a scripting language for games. Over the years the runtime had to be tweaked to provide a safe, secure sandboxed environment; we gradually started accumulating small library changes and tweaks.

Over the course of the last few years, instead of using Web-based stack for our player-facing application, Lua-based in-game UI and Qt-based editor UI, we've started consolidating a lot of the efforts and developing all of these using Roblox engine and Lua as a scripting language.

Having grown a substantial internal codebase that needed to be correct and performant, and with the focus shifting a bit from novice game developers to professional studios building games on Roblox and our own teams of engineers building applications, there was a need to improve performance and quality of the code we were writing.

Unlike mainline Lua, we also could not afford to do major breaking changes to the language (hence the 5.1 language baseline that remained unchanged for more than a decade). While faster implementations of Lua 5.1 like LuaJIT were available, they didn't meet our needs in terms of portability, ease of change and they didn't address the problem of developing robust code at scale.

All of these motivated us to start reshaping Lua 5.1 that we started from into a new, derivative language that we call Luau. Our focus is on making the language more performant and feature-rich, and make it easier to write robust code through a combination of linting and type checking using a gradual type system.

Complete rewrite?

A very large part of Luau codebase is written from scratch. We needed a set of tools to be able to write language analysis tools; Lua has a parser that is integrated with the bytecode compiler, which makes it unsuitable for complex semantic analysis. For bytecode compilation, while a single pass compiler can deliver better compilation throughput and be simpler than a full frontend/backend, it significantly limits the optimizations that can be done at the bytecode level.

Luau compiler and analysis tools are thus written from scratch, closely following the syntax and semantics of Lua. Our compiler is not single-pass, and instead relies on a set of analysis passes that run over the AST to produce efficient bytecode, followed by some post-process optimizations.

As for the runtime, we had to rewrite the interpreter from scratch to get substantially faster performance; using a combination of techniques pioneered by LuaJIT and custom optimizations that are able to improve performance by taking control over the entire stack (language, compiler, interpreter, virtual machine), we're able to get close to LuaJIT interpreter performance while using C as an implementation language.

The garbage collector and the core libraries represent more of an incremental change, where we used Lua 5.1 as a baseline but we're continuing to rewrite these as well with performance in mind.

While Luau doesn't currently implement JIT/AOT, this is likely to happen at some point; beyond the usual implementation challenges and security concerns, one significant limitation is that we don't have access to JIT on many platforms so for us maintaining excellent interpreted performance for gameplay and application code is more important than reaching peak FLOPS on numerical code.