# Package 'trampoline'

January 4, 2022

**Title** Make Functions that Can Recurse Infinitely

**Version** 0.1.1

**Description** Implements a trampoline algorithm for R that let's users write recursive functions that get around R's stack call limitations, enabling theoretically infinite recursion. The algorithm is based around generator function as implemented in the 'coro' package, and is based almost completely on the 'trampoline' module from Python <https://gitlab.com/ferreum/trampoline>.

**License** MIT + file LICENSE

**Encoding** UTF-8

**RoxygenNote** 7.1.2

**Imports** coro, fastmap, rlang (>= 0.1.2)

**Suggests** rmarkdown, knitr, bench, testthat (>= 3.0.0), roxygen2

**VignetteBuilder** knitr

**URL** https://github.com/rdinnager/trampoline, https://rdinnager.github.io/trampoline/

**Config/testthat/edition** 3

**NeedsCompilation** no

**Author** Russell Dinnage [aut, cre, cph] (<https://orcid.org/0000-0003-0846-2819>)

**Maintainer** Russell Dinnage <r.dinnage@gmail.com>

**Repository** CRAN

**Date/Publication** 2022-01-04 20:40:02 UTC

## R topics documented:

`trampoline` *Make a trampoline.*

## Description

This function takes a call to a generator factory, created by `coro::generator()` and runs it as a trampoline, which allows any recursion in the generator function to recurse theoretically forever (but usually just more than can be handled by R's default call stack limits).

## Usage

```
trampoline(call, ...)

tramampoline(call, ...)

trambopoline(call, ...)
```

## Arguments

| | |
|---|---|
| `call` | A call to a function or generator function. The function can be one defined already in the calling environment or higher or can be defined as an argument to `trampoline()`, see `...` argument. |
| `...` | A named list of functions or generator functions. Named arguments are function or generator function definitions where the name of the argument should be the desired name of the function (that is referred to also within the function for recursion, see examples to get a clearer idea of what this means). Passing multiple named arguments is possible and allows specification of functions that can be used within the generator function that is called in `call` (again, the examples might make this clearer). |

## Value

If `trm_return()` or `trm_tailcall()` is called within the recursive generator function, `trampoline()` will return the final return value from the final recursion. Otherwise it will return `NULL` invisibly (in case the recursion is only for its side-effects). See the examples for how this works.

## Examples

```
## standard recursive function exhausts stack:
print_numbers <- function(n) {
  if(n >= 1) {
    print_numbers(n - 1)
    print(n)
  }
}
try(print_numbers(5000))

## use trampoline with a coro generator instead
```

```
print_numbers <- coro::generator(function(n) {
  if(n >= 1) {
    yield(print_numbers(n - 1))
    print(n)
  }
})
nums <- capture.output(
  trampoline(print_numbers(5000))
)
cat(tail(nums))

## Or just use a plain function (but still use yield())
print_numbers <- function(n) {
  if(n >= 1) {
    yield(print_numbers(n - 1))
    print(n)
  }
}

trampoline(print_numbers(5))

## use an alias or another
tramampoline(print_numbers(5))
trambopoline(print_numbers(5))

## use multiple mutually recursive functions
even <- function(n) {
  if (n == 0) trm_return(TRUE) else yield(odd(n - 1))
}

odd <- function(n) {
  if (n == 0) trm_return(FALSE) else yield(even(n - 1))
}

## doesn't work (you must pass odd in because trampoline
## only converts first called function to generator by default)
try(trampoline(even(100)))

## does work
trampoline(even(100), odd = odd)

## you can specify your recursive function in the trampoline
## call if you want.
## Return a value using trm_return():
trampoline(factorial(13),
           factorial = function(n) {
             if(n <= 1) {
               return(trm_return(1))
             }
             val <- yield(factorial(n - 1))
             return(val * n)
           })
```

```
## convert to using tail call optimization by wrapping
## recursive call in trm_tailcall()
trampoline(factorial(13),
           factorial = function(n, x = 1) {
             force(x) ## necessary thanks to R's lazy evaluation
             if(n <= 1) {
               return(trm_return(x))
             }
             val <- trm_tailcall(factorial(n - 1, x * n))
             return(val)
           })
```

---

`trm_return` *Flag a return value*

---

## Description

Wrap a return value in your recursive function with `trm_return()` to have it passed along and returned by your final recursion.

## Usage

```
trm_return(x)
```

## Arguments

| | |
|---|---|
| `x` | A value to be returned at the end of all recursions |

## Value

`x` with added class attribute 'trampoline_return'

## Examples

```
trampoline(factorial(13),
           factorial = function(n) {
             if(n <= 1) {
               return(trm_return(1))
             }
             val <- yield(factorial(n - 1))
             return(val * n)
           })
```

`trm_tailcall` *Flag a tail call*

## Description

If you can specify your recursive function such that the recursive call is in 'tail position' (that is, the very last operation in your function), you can take advantage of tail call optimization. Just wrap your recursive call in `trm_tailcall()`

## Usage

```
trm_tailcall(x)
```

## Arguments

| | |
|---|---|
| `x` | A recursive call within generator fed to `trampoline()` |

## Value

`x` with added class attribute 'trampoline_tailcall'

## Examples

```
trampoline(factorial(13),
           factorial = function(n, x = 1) {
             force(x) ## necessary thanks to R's lazy evaluation
             if(n <= 1) {
               return(trm_return(x))
             }
             val <- trm_tailcall(factorial(n - 1, x * n))
             return(val)
           })
```

# Index