TracePoint

Examining the execution flow of your code

Create your tracer

Create your tracer

```
trace = TracePoint.new(:call, :c_call) do |tp|

end
```

Create your tracer

```
methods = Set.new
trace = TracePoint.new(:call, :c_call) do |tp|
  methods << tp.method_id
end
```

:call, :return

Call / return from a Ruby function

:c_call, :c_return

Call / return from a C function

:b_call, :b_return

Call / return from a block

:class, :end

Start / end a class/module definition

:raise

Raise an exception

:line

Execute a line of Ruby code

Tracepoint#event

Name of the tracepoint event

Tracepoint#method_id

Underlying method name

Tracepoint#defined_class

Class that defined the method

Tracepoint#self

The class at runtime

Tracepoint#path

Path of source file of execution point

Tracepoint#return_value

Value returned from functions/blocks

Trace your code

```ruby
methods = Set.new
trace = TracePoint.new(:call, :c_call) do |tp|
  methods << tp.method_id
end
```

Trace your code

```
methods = Set.new
trace = TracePoint.new(:call, :c_call) do |tp|
  methods << tp.method_id
end

trace.enable { func2 }
puts methods
```

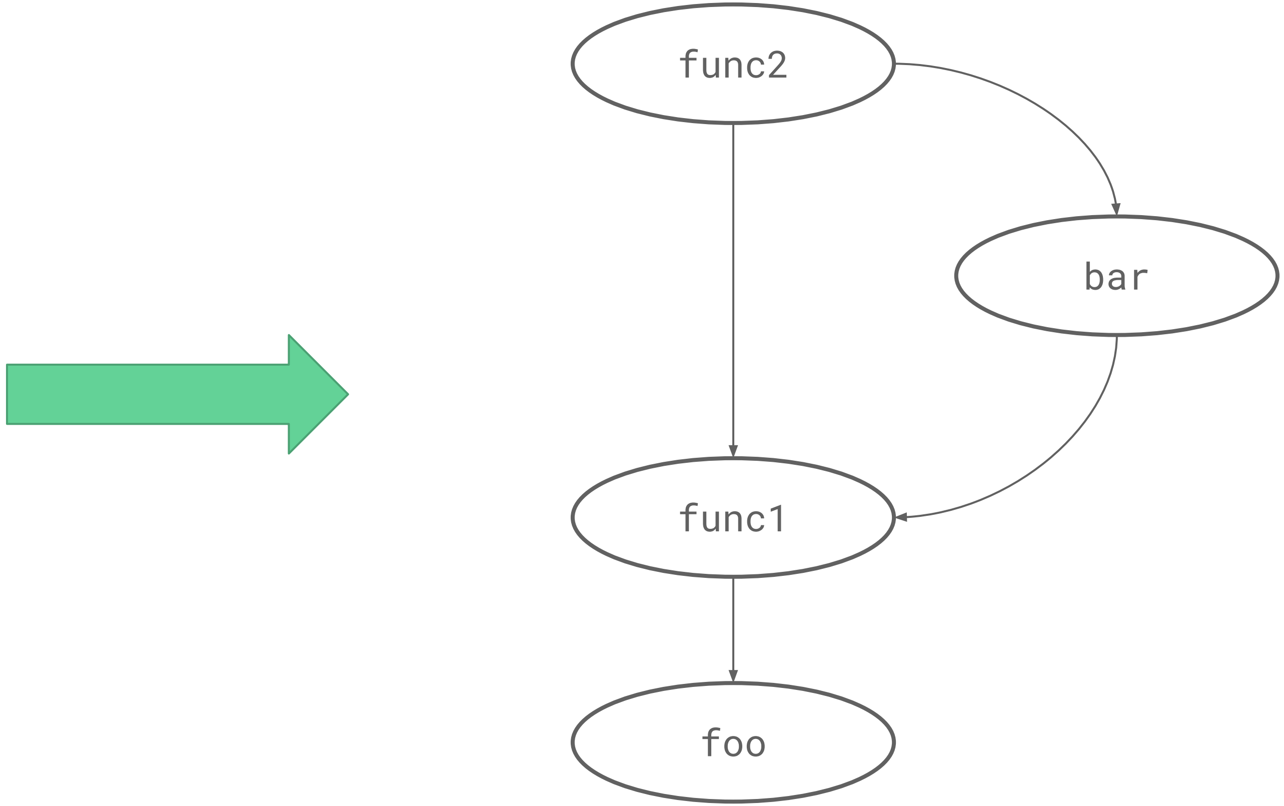
Trace your code

```
def foo
  puts “test”
end

def func1
  foo
end

def bar
  func1
end
```

```
def func2
  func1
  bar
end

func2
```

```
func2
func1
foo
puts
write
bar
```

```ruby
def foo
  sleep 0.1
end

def func1
  foo
end

def bar
  func1
end

def func2
  func1
  bar
end
```



Call graphs

TracePoint doesn’t track your call stack

```ruby
def foo
  sleep 0.1
end

def func1
  foo
end

def bar
  func1
end

def func2
  func1
  bar
end

func2
```

```ruby
trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call





  when :return

  end
end
```

```ruby
def foo
  sleep 0.1
end

def func1
  foo
end

def bar
  func1
end

def func2
  func1
  bar
end

func2
```

:call	:func2
:call	:func1
:call	:foo
:return	:foo
:return	:func1
:call	:bar
:call	:func1
:call	:foo
:return	:foo
:return	:func1
:return	:bar
:return	:func2

```ruby
stack = []

trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call
    stack << tp.method_id





  when :return
    stack.pop
  end
end
```

```ruby
def foo
  sleep 0.1
end

def func1
  foo
end

def bar
  func1
end

def func2
  func1
  bar
end

func2
```

:call	:func2	[:func2]
:call	:func1	[:func2, :func1]
:call	:foo	[:func2, :func1, :foo]
:return	:foo	[:func2, :func1]
:return	:func1	[:func2]
:call	:bar	[:func2, :bar]
:call	:func1	[:func2, :bar, :func1]
:call	:foo	[:func2, :bar, :func1, :foo]
:return	:foo	[:func2, :bar, :func1]
:return	:func1	[:func2, :bar]
:return	:bar	[:func2]
:return	:func2	[]

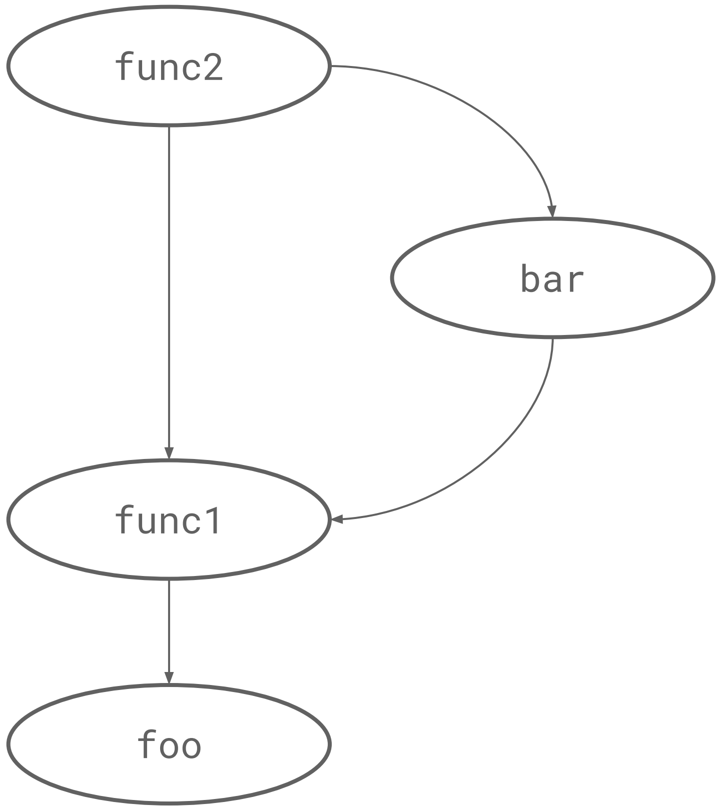
```ruby
def foo
  sleep 0.1
end

def func1
  foo
end

def bar
  func1
end

def func2
  func1
  bar
end

func2
```

:call	:func2	[:func2]
:call	:func1	[:func2, :func1]
:call	:foo	[:func2, :func1, :foo]
:return	:foo	[:func2, :func1]
:return	:func1	[:func2]
:call	:bar	[:func2, :bar]
:call	:func1	[:func2, :bar, :func1]
:call	:foo	[:func2, :bar, :func1, :foo]
:return	:foo	[:func2, :bar, :func1]
:return	:func1	[:func2, :bar]
:return	:bar	[:func2]
:return	:func2	[]

```ruby
stack = []

trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call
    stack << tp.method_id





  when :return
    stack.pop
  end
end
```

```
stack = []
callgraph = {}
trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call
    stack << tp.method_id
    if stack.length >= 2
      callgraph[stack[-2]] ||= Set.new
      callgraph[stack[-2]] << tp.method_id
    end
  when :return
    stack.pop
  end
end
```

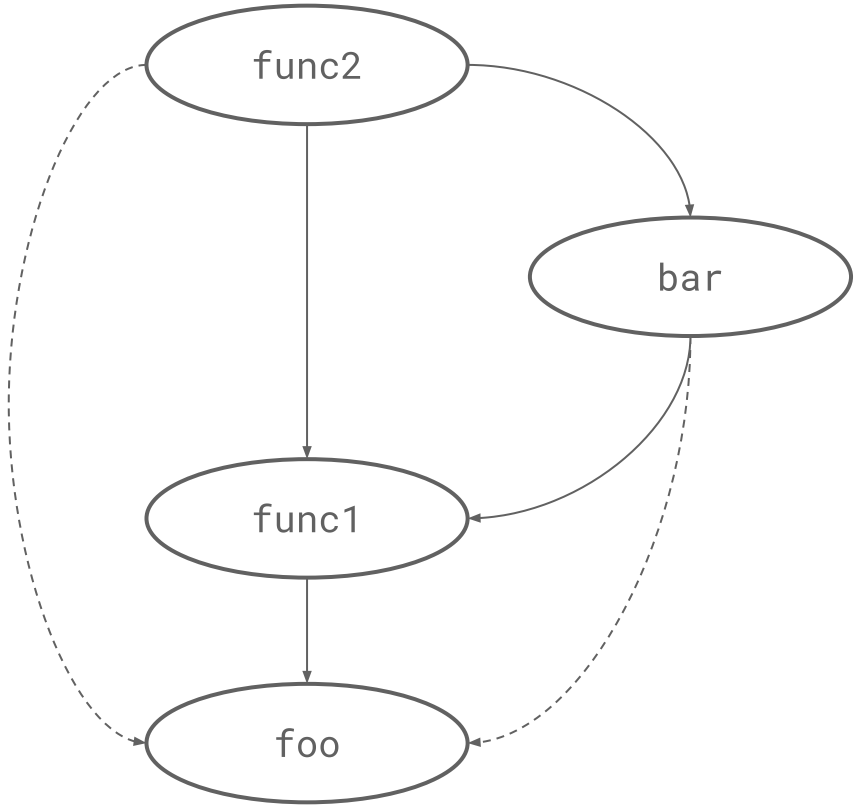
```
{
  func1: #<Set: {:foo}>,
  bar:   #<Set: {:func1}>,
  func2: #<Set: {:func1, :bar}>,
}
```


$$\begin{tikzcd}
\text{func2} \arrow[d] \arrow[r, bend left] & \text{bar} \arrow[dl, bend left] \\
\text{func1} \arrow[d] & \\
\text{foo} &
\end{tikzcd}$$

There’s still more to do…

Include self

Filter gems



Compute the transitive closure

```ruby
TracePoint.trace(:raise) do |tp|
  exception = tp.raised_exception.inspect
  location = "#{tp.path}:#{tp.lineno}"
  puts "Raised #{exception} at #{location}"
end
```

```ruby
coverage = {}
trace = TracePoint.new(:line) do |tp|
  coverage[tp.path] ||= Set.new
  coverage[tp.path] << tp.lineno
end
```

```ruby
stack = []
profile = Hash.new(0)
trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call
    stack << [tp.method_id, Time.now]
  when :return
    method, start_time = stack.pop
    profile[method] += Time.now - start_time
  end
end
```

```ruby
stack = []
tests_to_run = {}
trace = TracePoint.new(:call, :return) do |tp|
  case tp.event
  when :call
    stack << [tp.method_id, tp.path]
  when :return
    _, path = stack.pop
    tests_to_run[path] ||= Set.new
    tests_to_run[path] << stack.first unless stack.empty?
  end
end
```

Thanks!

thegedge

thegedge