Ruby master - Feature #12543

explicit tail call syntax: foo() then return

07/02/2016 05:24 PM - mame (Yusuke Endoh)

Status: Assigned
Priority: Normal
Assignee: matz (Yukihiro Matsumoto)
Target version:

Description
How about introducing a new syntax for tail call?

```ruby
def foo()
  foo()
end
foo() #=> stack level too deep

def bar()
  bar() then return
end
bar() #=> infinite loop
```

- no new keyword (cf. goto foo())
- no conflict with any existing syntax
- an experimental patch is available (attached)
- no shift/reduce nor reduce/reduce conflict in parse.y

--
Yusuke Endoh mame@ruby-lang.org

Related issues:
Related to Ruby master - Feature #6602: Tail call optimization: enable by def...
Has duplicate Ruby master - Feature #16945: Enable TCO by use of special form

Feedback
Has duplicate Ruby master - Feature #16945: Enable TCO by use of special form

Closed

History
#1 - 07/02/2016 05:24 PM - mame (Yusuke Endoh)
- Subject changed from explicit tail call syntax: foo() then return to explicit tail call syntax: foo() then return

#2 - 07/02/2016 05:25 PM - mame (Yusuke Endoh)
- Related to Feature #6602: Tail call optimization: enable by default? added

#3 - 07/19/2016 06:53 AM - ko1 (Koichi Sasada)
mame-san:
Do you have use-cases?

#4 - 07/19/2016 09:10 AM - matz (Yukihiro Matsumoto)
- Status changed from Open to Closed

I am not positive. This may not work under tracing. I am for adding tail-call optimization, but Koichi do not love the idea either.

Matz.

#5 - 09/02/2019 03:29 AM - ko1 (Koichi Sasada)
- Status changed from Closed to Assigned

Another idea: tailcall return foo()

- Background: return is void expression and any method(return) is not prohibited on current compiler. So nobody use it == no-incompatibility. Any keywords are acceptable (example: goto return foo()).
I like Endoh-san's original idea, tailcall explicitly. It is similar to goto, and programmer should understand there is no backtrace.

I heard that someone want to use taicall with pattern matching which will be introduced into Ruby 2.7.

#6 - 09/02/2019 07:06 AM - duerst (Martin Dürst)

I don't think tail call optimization should be a feature that is switched on or off by the programmer at each location. I think it should be an option used on execution, and it should be ON by default.

We want programs to be fast, and tail call optimization makes them faster. In #6602, there was the opinion that tail calls are rare in Ruby, but that may also have to do with the fact that they are not optimized. So to some extent, it's a chicken and egg problem.

What usually happens is that users write programs and run them. If they run faster, that's good. That's why I think tail call optimization should be on by default. What happens next is that occasionally, there's a bug. That bug may produce a stack trace. The stack trace should include a hint as to where tail call optimization was in effect. The programmer will read the stack trace, and if they suspect that the bug is somewhere near the tail call, they can run the program with tail calls switched off by using an option.

For me, having tail calls off by default, or having syntax to switch them on per calling location seems to put the chart before the horse. I hope this can be avoided.

#7 - 09/02/2019 07:51 AM - mame (Yusuke Endoh)

I'm strongly against "ON by default". It makes the backtrace difficult to understand. Consider the following program:

```
1: def foo
2:   raise
3: end
4:
5: def bar
6:   foo
7: end
8:
9: bar
```

If tail-call optimization is used by default, it will print:

```
Traceback (most recent call last):
  1: from test.rb:9:in `<main>'
  2: test.rb:2:in `foo': unhandled exception
```

The frame of bar is removed due to tail-call optimization, so the debugger must guess how it reached at Line 2 from Line 9.

This issue would be incredibly difficult when multiple frames are omitted. It would be not so rare on practical programs. I believe that "easy to debug" is one of the most important properties in Ruby.

#8 - 09/02/2019 08:37 AM - duerst (Martin Dürst)

mame (Yusuke Endoh) wrote:

I'm strongly against "ON by default". It makes the backtrace difficult to understand. Consider the following program:

```
1: def foo
2:   raise
3: end
4:
5: def bar
6:   foo
7: end
8:
9: bar
```

If tail-call optimization is used by default, it will print:

```
Traceback (most recent call last):
  1: from test.rb:9:in `<main>'
  2: test.rb:2:in `foo': unhandled exception
```

This should be changed to something like:

```
Traceback (most recent call last):
  1: from test.rb:9:in `<main>'
  2: test.rb:2:in `foo': unhandled exception
[...]
  1: from test.rb:9:in `<main>'
  2: test.rb:2:in `foo': unhandled exception
```

Of course, the exact name of the exception and the wording of the message can still be improved. Implementation should be easy, just set a flag on the stack frame above the one that is eliminated by the tail call optimization.

The frame of bar is removed due to tail-call optimization, so the debugger must guess how it reached at Line 2 from Line 9.

Guessing is of course not prohibited, but better use the option to get the full trace.
This issue would be incredibly difficult when multiple frames are omitted. It would be not so rare on practical programs. I believe that "easy to debug" is one of the most important properties in Ruby.

I agree that "easy to debug" is important for Ruby. But I don't think my proposal makes debugging very difficult.

### #9 - 09/02/2019 02:00 PM - mame (Yusuke Endoh)

I don't like --tail-call-optimization-off. I will not specify the option, see an omitted backtrace, and then I must re-run my code with the option. It is not an easy-to-debug language for me.

And I have another concern. If tail call optimization is on by default, some people will strongly depend on it. For example, someone may write:

```ruby
def main_loop
  socket = server_socket.accept
  ...
  main_loop
end
```

This code will break when --tail-call-optimization-off is specified.

So, I think it should be off by default, and it would be good to allow to write:

```ruby
def main_loop
  socket = server_socket.accept
  ...
  return and main_loop # explicit tail call
end
```

### #10 - 09/04/2019 02:54 PM - Dan0042 (Daniel DeLorme)

Questions:

1. Is it possible to use "partial" tail-call optimization, where the backtrace is kept but all other frame state is discarded?
2. Is it possible to detect tail-recursion and turn on optimization just for that?

### #11 - 09/07/2019 08:24 AM - shyouhei (Shyouhei Urabe)

Dan0042 (Daniel DeLorme) wrote:

Questions:

1. Is it possible to use "partial" tail-call optimization, where the backtrace is kept but all other frame state is discarded?

That's heavier than a normal method call; we don't "keep" a Thread::Backtrace::Location now. Instances of that class are constructed on-the-fly when necessary. However if we do a "partial" optimization like you say we have to explicitly keep them, which adds extra overhead every time when you call something -- not only when backtraces are needed.

1. Is it possible to detect tail-recursion and turn on optimization just for that?

That's what's requested in this request. "Turn optimization just for this return" is what's called then return here.

### #12 - 09/07/2019 09:57 AM - Eregon (Benoit Daloze)

duerst (Martin Dürst) wrote:

We want programs to be fast, and tail call optimization makes them faster.

That's not true in general. It might make recursive methods faster, but it makes normal methods calls that happen at the end of a method body slower with a JIT, because the TCO call is a loop calling two different methods, instead of straight line code from inlining the method call which happens to be in tail position.

Is it possible to use "partial" tail-call optimization, where the backtrace is kept but all other frame state is discarded?

I think not in general, because then we'd need a stack for the backtrace and TCO no longer removes the need for stack space (it might be feasible to, e.g., keep a counter if it's a self-recursive call but not in general).

I strongly agree that if we add TCO, it should be explicit in the code, otherwise it breaks backtraces/tracing/debugging and it slows downs JIT-ed execution.

Maybe it could be implicit if it's restricted to self-recursive calls (e.g.; def bar; ...; bar; end).
In that case, there is not much information in the backtrace to be lost, and the performance of JIT-ed execution is likely similar, while allowing the self-recursive style (instead of a loop).

I'm not sure how useful the self-recursive style is in Ruby though. I am thinking the example above with `main_loop` is more readable and easier to understand what it actually does with a while socket = server_socket.accept loop.

Do we have motivating examples for this feature?

#13 - 06/10/2020 11:58 AM - nobu (Nobuyoshi Nakada)
- Has duplicate Feature #16945: Enable TCO by use of special form added

#14 - 06/13/2020 01:27 AM - dsisneros (Dominic Sisneros)
+1 for tail call optimization - either explicit or automatic

#15 - 04/18/2021 03:02 PM - jwmittag (Jörg W Mittag)

mame (Yusuke Endoh) wrote in #note-9:

And I have another concern. If tail call optimization is on by default, some people will strongly depend on it.

That's the point. Proper Tail Calls allow you to write code that is otherwise impossible to write. But that is crucially dependent on the knowledge that the call will always be optimized, no matter what.

For example, you can very elegantly express State Machines in an Object-Oriented manner: every state is an object, every transition is a method call. But in doing this, you just move from state to state through the state machine, you never return. This very elegant Object-Oriented encoding of State Machines depends on Proper Tail Calls.

Guy L. Steele has argued that Proper Tail Calls are required for Object-Orientation Languages.

One thing we should be very careful about is whether we are talking about Language Semantics (these are usually called Proper Tail Calls or Properly-Implemented Tail Call Handling (PITCH)) or a simple Compiler Optimization (these are typically Tail Call Optimization). The main difference is that Proper Tail Calls are a guarantee made by the language specification. These calls are guaranteed to be optimized, under any circumstances, in any implementation (YARV, TruffleRuby, JRuby, Opal, Artichoke, …). Whereas TCO may or may not be applied depending on the implementation, the version, the optimization level, the surrounding code, the phase of the moon, a command line option, or even a random coin flip.

Proper Tail Calls are required for certain kinds of modularity, they are required for certain kinds of designs. When I talk about Proper Tail Calls, that is what I mean: IFF a call meets the definition of a Tail Call (whatever definition the community settles on), then it is guaranteed to be optimized in every Ruby implementation, always.

Only this kind of guarantee will allow one to write code that depends on Proper Tail Calls.

Files

then_return.patch 9.18 KB 07/02/2016 mame (Yusuke Endoh)