Hi,

Some hours ago, Matz proposed turning on "tail call optimization" by default from Ruby 2.0.

What do you think about it?

Background

Tail call: Method invocation at last of method body.

Tail call optimization: Eliminating the new stack frame creation when method invocation is "tail call".

For example, the method bar() is located at last of method foo(), so bar() is "tail call".

```ruby
def foo()
  ...
  bar()
end
```

In this case, after invocation of method bar(), foo()'s method frame information (which contains local variables, program counter, stack pointer and so on) is no longer needed because method foo() doesn't work after that (correctly, method foo() only does "return").

Next example, a simple recursion code by foo(). Of course, foo() is "tail call".

```ruby
def foo()
  ...
  foo()
end
```

Current Ruby causes stack overflow error because such recursion consumes the (VM) stack. However, using tail call optimization, VM doesn't consume stack frame any more.

Such recursion can be converted to simple loop:

```ruby
def foo
  while true
    foo()
  end
end
```

Someone calls tail-call opt as "tail recursion optimization" because recursion is famous use-case (*1).

*1: Generally, tail-recursion optimization includes another optimization technique - "call" to "jump" translation. In my opinion, it is difficult to apply this optimization because recognizing "recursion" is difficult in Ruby's world.

Next example. fact() method invocation in "else" clause is not a "tail call".

```ruby
def fact(n)
  if n < 2
    1
  else
    n * fact(n-1)
end
```
If you want to use tail-call optimization on `fact()` method, you need to change `fact()` method as follows (continuation passing style).

```ruby
def fact(n, r)
  if n < 2
    r
  else
    fact(n-1, n*r)
  end
end
```

In this case, `fact()` is tail-call (and a bit difficult to read/write).

Of course, the following code is easy to understand and short.

```ruby
(1..n).inject(:*)
```

Last examples. Recognizing tail-call is a bit difficult.

```ruby
def foo
  begin
    bar2() # not a tail-call
    rescue
      bar3() # not a tail-call
      rescue
        bar4() # not a tail-call
        ensure
          bar5() # tail-call!
    end
  end
end
def foo
  while true
    return bar("break") # tail-call? (current CRuby can't handle "break" in eval().
  end
end
```

CRuby 1.9 has a code tail-call optimization (not tested yet. maybe there are several bugs). However, it is off by default because of several problems described in next section.

**Problems:**

- (1) backtrace: Eliminating method frame means eliminating backtrace.
- (2) `set_trace_func()`: It is difficult to probe "return" event for tail-call methods.
- (3) semantics: It is difficult to define tail-call in document (half is joking, but half is serious)

References:

- [ruby-core:20273]
- [ruby-core:20307]
- [ruby-core:22736]
- [ruby-core:22790]

Maybe (1) has big impact for ordinal users.

For example:

```ruby
def foo
  bar()
end

def bar
  baz()
end
```
def baz
  raise("somethig error")
end

In this case, backtrace information only include "baz", because bar() in foo and baz() in bar are "tail-call". Users can't see eliminated frame information in backtrace.

This is why we don't introduce them by default to Ruby 1.9.

Discussion

Many people ask us that "why don't you introduce tail-call optimization? it is very easy technique." I wrote reasons above.

Matz said "it seems small impact enough. Go ahead". (I doubt it ;P )

Yusuke Endo proposed that introducing special form (for example, send_tail(:foo, ...)) to declare tail call. Users only use this special form when the backtrace information can be eliminated (*2).

(*2) Special form "goto foo()" is nice joking feature :) I like it but I believe Matz will reject it.

Akira Tanaka introduced that special backtrace notation like:

```ruby
baz
... (eliminated by tail call optimization)
main
```

to represent eliminating method invocation information. We can know they were eliminated (good) but we can't know what method frames were eliminated (bad).

Conclusion

Matz wanted to introduce it. However it has several problems. Should we turn on this optimization by default?

Sorry for long (and poor English) article. Comments and proposals are welcome (with short English, long Ruby codes ;p).

Thanks,
Koichi

Related issues:
Related to Ruby master - Feature #12543: explicit tail call syntax: foo() the...

Assigned

History

#1 - 06/18/2012 06:10 PM - nobu (Nobuyoshi Nakada)
- Description updated

#2 - 06/18/2012 08:29 PM - trans (Thomas Sawyer)
I'd rather have the backtrace information and depend on Moore's law for "enhancements".

However, in the the case of tail-recursion, it could be ok since the stack entry is essentially a facsimile of the previous.

#3 - 06/18/2012 09:53 PM - ko1 (Koichi Sasada)
(2012/06/18 20:29), trans (Thomas Sawyer) wrote:

  However, in the the case of tail-recursion, it could be ok since the stack entry is essentially a facsimile of the previous.

Do you mean the following code isn't applied tail call optimization?

```ruby
# double recursion (foo -> bar -> foo -> bar -> ...)
def foo
  bar()
def bar()
end
def bar()
```
foo()
end

I agree it is one option for this problem.

--
// SASADA Koichi at atdot dot net

#4 - 07/14/2012 06:36 PM - mame (Yusuke Endoh)
  - Status changed from Open to Assigned

#5 - 10/10/2012 12:36 PM - headius (Charles Nutter)
FWIW, JRuby will not be able to support TCO until the JVM supports TCO, so it won't work across implementations. I don't say that to hold back progress...just stating facts.

#6 - 10/11/2012 06:12 PM - alexeymuranov (Alexey Muranov)
  +1 for goto foo()
Tail call optimization looks to me like a tamed form of goto.

#7 - 10/27/2012 07:07 AM - ko1 (Koichi Sasada)
  - Target version changed from 2.0.0 to 2.6
Alexey: to introduce new syntax (or method), we need more discussion (mainly on name ;)).
I changed target to next minor.

#8 - 11/24/2012 03:09 PM - mame (Yusuke Endoh)
  - Assignee changed from mame (Yusuke Endoh) to ko1 (Koichi Sasada)

#9 - 07/02/2016 05:25 PM - mame (Yusuke Endoh)
  - Related to Feature #12543: explicit tail call syntax: foo() then return added

#10 - 01/31/2017 09:42 AM - ko1 (Koichi Sasada)
  - Status changed from Assigned to Feedback
  - Description updated
I hope someone propose smart answers.

#11 - 02/01/2017 02:36 PM - Eregon (Benoit Daloze)
I think losing the backtrace is fairly bad in a mostly-imperative language.
Tail calls would allow tail-recursion, but this is not very frequent in Ruby.

In my opinion, recursion becomes really interesting when you can invoke it from different points, like walking over a tree, but this is not tail-recursive anymore.

#12 - 02/02/2017 12:51 PM - mame (Yusuke Endoh)
Benoit Daloze wrote:
  I think losing the backtrace is fairly bad in a mostly-imperative language.
  Tail calls would allow tail-recursion, but this is not very frequent in Ruby.

Agreed. There are few cases in Ruby where tail-call optimization is really useful. I still believe that any explicit syntax to mark such a case (#12543) is a good way.

#13 - 12/25/2017 06:15 PM - naruse (Yui NARUSE)
  - Target version deleted (2.6)