

延续(continuation)

及在编译器优化中的应用

@右席

pwq1989@gmail.com

内容包括：

- 延续(continuation)及相关的概念
- CPS及其在编译器中的应用
- 在解释器中使用CPS实现尾递归优化
- 处理更加复杂的情况

延续(continuation)

- 可以简单的理解为剩下的计算，例如：

```
(set! a (+ b 2))
```



```
(lambda(k) (set! a k))
```

Continuation的提供

1. First class continuation

语言级别原生支持，例如：

Scheme中的call/cc函数 (call with current continuation)

tinymoe中的continuation关键字

2. CPS (continuation passing style)

将continuation作为参数在所有函数间传递的程序风格：

```
function foo(a, k) {  
    k(a); // equal to "return a"  
}
```


Continuation与程序结构

作为控制流来说，是非常强大的。

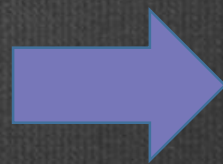
- Exception
- Repeat
- Iterator
- Coroutine
- ...

Coroutine的小栗子

准备工作：call/cc in Scheme

call/cc(call with current continuation)是一个函数，它接受一个有一个参数的函数作为参数，然后把continuation传递给它的参数（就是那个函数），执行它~

```
(+ 2  
  (call/cc  
    (lambda (k) (k 2))))
```



Output: 4

Coroutine的小栗子(续)

它的输出是什么？

```
(define r1
  (lambda (cont)
    (display "I'm in r1!")
    (newline)
    (r1 (call/cc cont))))

(define r2
  (lambda (cont2)
    (display "I'm in r2!")
    (newline)
    (r2 (call/cc cont2))))

(r1 r2)
```

Coroutine的小栗子(续)

它的输出是什么？

```
I'm in r1!  
I'm in r2!  
I'm in r1!  
I'm in r2!  
I'm in r1!  
I'm in r2!  
...
```

```
(define r1  
  (lambda (cont)  
    (display "I'm in r1!")  
    (newline)  
    (r1 (call/cc cont))))  
(define r2  
  (lambda (cont2)  
    (display "I'm in r2!")  
    (newline)  
    (r2 (call/cc cont2))))  
(r1 r2)
```

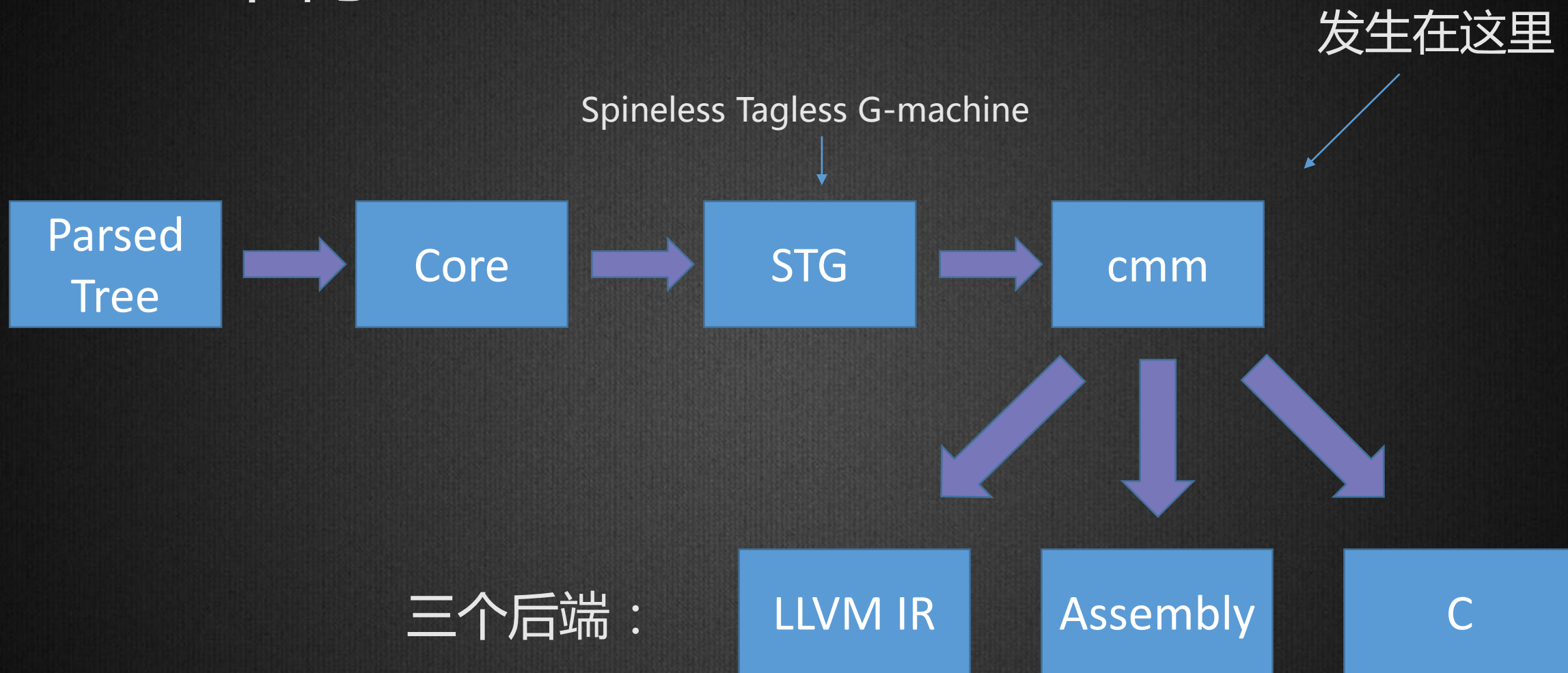

CPS作为IR(intermediate representation)

大多应用在一些偏函数式的语言编译器实现中。

真实世界中有：

1. GHC(Haskell编译器)的cmm优化的一个pass
2. Mlton(SML编译器，现已改用SSA =。=)
3. 一些Scheme编译器(Chicken, Rabbit, ORBIT etc.)
4. Tnymoe(一个允许在程序控制continuation的语言)

GHC架构



GHC中的CPS Pass

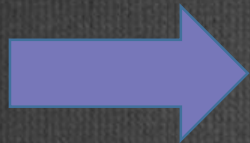
对Cmm(C--)进行优化，该阶段涉及的方面有：

- Proc-Point Analysis
- Calling Conventions
- Live Value Analysis
- Stack Layout

GHC中的CPS Pass

```
f {  
  y = 1;  
  z = 2;  
  x = call g(a, b);  
  return x+y+z;  
}
```


Cmm code



```
f {  
  y = 1;  
  z = 2;  
  push_continuation h [y, z];  
  jump g(a, b);  
}  
  
foreign "ret" h(x) {  
  (y, z) = expand_continuation;  
  return x+y+z;  
}
```

Cmm code

Tinymoe中的continuation

Tinymoe(作者:  陈梓瀚 vczh)是一个命令式语言, 支持在程序中获取指定代码块或表达式的continuation, 便可以在语言中以类库的形式做出 try...catch... coroutine等。

```
phrase main
```

```
  try
```

```
    set a to 1
```

```
    set b to 0
```

```
    set c = a / b
```

raise exception: 实现见下一页

作为continuation, 取出传递过来的exception

```
  catch the exception
```

```
    print the exception
```

```
  end
```

```
end
```

Tinymoe中的continuation

获取raise (...) 这类语句的continuation

```
cps (state) (continuation)
sentence raise (exception)
    reset continuation state state to raising exception
    set field argument of state to exception
    fall into the previous trap
end
```

标记在状态中，然后退回到上一步的continuation中

CPS在解释器中的应用

解释器(Interpreter)的特点就是按句翻译，一行一行来，一般不会改变程序本身的结构，结构简单，跑起来慢。

后面的例子会演示以CPS风格对程序进行解释，并实现尾递归优化 (TCO)。

(附) 代码地址：<https://gist.github.com/pwq1989/9554c6f5f8fea0afcc01> 感谢@believe

尾递归优化(TCO)的实现

例子中会对以下代码解释执行。begin标明程序块的开始，set表示变量赋值，if就是if，lambda标明是函数，[]中第一个是函数，后面为函数调用时候需要的参数。

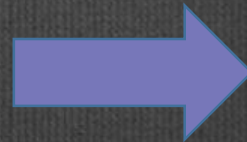
```
['begin',  
  ['set', 'sum', ['lambda', ['low', 'hi', 'acc'],  
    ['if', ['<', 'low', 'hi'],  
      ['sum', ['+', 'low', 1], 'hi', ['+', 'low', 'acc']],  
      ['+', 'acc', 'low']]]],  
  ['sum', 0, 1000, 0]  
]
```


尾递归优化(TCO)的实现

如果没有TCO，逐句解释，栈就一层一层的叠了下去。对于实现上没有TCO的语言来说，调用栈一多就爆栈了



```
function foo(x) {  
    if (x > 100) return 100;  
    return foo(x + 1);  
}
```



```
foo...  
    foo...  
        foo...  
            foo...  
                foo...  
                    foo...  
next...
```

尾递归优化(TCO)的实现

关键部分：每次执行的时候传入continuation

```
interpret(root, e, k);  
interpretE(expression, e, k);
```

`['set', 'sum', ['+', 1, 2]]`

传入最外面的k，set是内置函数
添加第一个参数 (sum) 作为symbol到
Environment中

继续执行interpret(code, e, k2)，最后结果
以k2(return_value)的形式返回

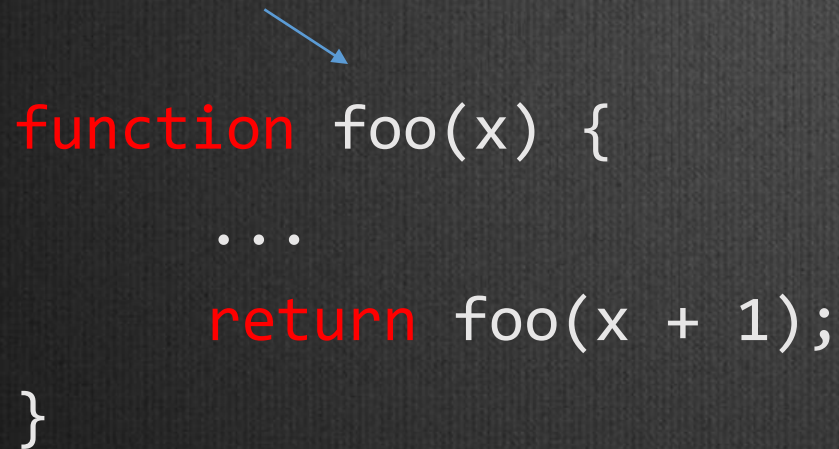
传入的k是interpretE(code, e, k1)

尾递归优化(TCO)的实现(续)

关键步骤：evaluate()函数中

```
while(k is not empty && k is returning) {k = k->origin}
```

标记为会被递归的函数(特殊对待)，在调用
时候会备注，并标注上一级continuation



```
function foo(x) {  
    ...  
    return foo(x + 1);  
}
```


The diagram shows a function definition for 'foo(x)'. An arrow points from the text '标记为会被递归的函数(特殊对待)' to the 'function' keyword. Another arrow points from the text '并标注上一级continuation' to the 'return foo(x + 1);' line.

执行(evaluate)的时候，会检查是否是尾递归，直接返回到最初的continuation

如何应对更复杂的情况

下面的代码也应该属于尾递归，不过前面的解释器是无法处理的。

```
function foo(x) {  
    ...  
    return 6 + foo(x + 1);  
}
```



foo(x+1)的k为 (+ 6)，而不是foo，无法直接跳转回去

如何应对更复杂的情况(续)

如果对代码进行转换。改写为cps的形式，在参数中调用 continuation

```
function foo2(x, k) {  
    ...  
    return foo2(x + 1, function(r){return k(6 + r)});  
}
```

如何应对更复杂的情况(续)

如果对代码进行转换。改写为cps的形式，在参数中调用 continuation

```
function foo2(x, k) {  
    ...  
    return foo2(x + 1, function(r){return k(6 + r)});  
}
```

也是有问题的。。。

因为在套满continuation的回调中的call stack也很深

如何应对更复杂的情况(续)

解决方案：改写代码为

```
function foo2(x, acc) {  
    ...  
    return foo2(x + 1, 6 + acc);  
}
```

```
function foo(x) {  
    return foo2(x, 0);  
}
```



为什么是0

如何应对更复杂的情况(续)

0是(+)所对应的单位元(幺元, Identity)

$$0 + 6 = 6 + 0 = 6$$

如果编译器要处理这个情景,就必须能识别出那一系列函数是否提供了Monoid(幺半群)的规范(接口)

Haskell在标准库中所提供Monoid如下:

```
class Monoid a where
    mempty  :: a
    mappend :: a -> a -> a
    ...
```


如何应对更复杂的情况(续)

所以目标函数只要实现了Moniod接口，Haskell是有能力(?)在编译器上做到上述优化的

```
newtype Sum a = Sum { getSum :: a }
```

```
instance Num a => Monoid (Sum a) where  
    mempty = Sum 0  
    mappend (Sum x) (Sum y) = Sum (x + y)
```

致 谢

感谢所有帮助指导过我的人