Lecture 5

Implementing Coroutines

compile AST to bytecode, bytecode interpreter

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!
CS164: Introduction to Programming Languages and Compilers, Spring 2012
UC Berkeley

What you will learn today

Implement Asymmetric Coroutines

- why a recursive interpreter with implicit stack won't do

Compiling AST to bytecode

- this is your first compiler; compiles AST to "flat" code

Bytecode Interpreter

- bytecode can be interpreted without recursion
- hence no need to keep interpreter state on the call stack

PA2

PA2 was released today, due Sunday

- bytecode interpreter of coroutines
- after PA2, you will be able to implement iterators
- in PA3, you will build Prolog on top of your coroutines
- extra credit: cool use of coroutines (see L4 reading)

HW3 has been assigned: cool uses of coroutines

This homework is not graded. While it is optional, you are expected to know the homework material:

1) lazy list concatenation
2) regexes

Solve at least the lazy list problem before you start on PA2

Code 4 Cal Hackathon

Friday, February 3, 9:00pm to Saturday, February 4 5:00pm

Soda Hall Wozniak Lounge

Grand Prize: $1,500!!!

Register @ http://code4cal.eventbrite.com/

More information about the event at http://stc.berkeley.edu.

The Student Technology Council (STC), an advisory council for the UC Berkeley CIO, Shel Waggener, is hosting Code 4 Cal, **a hackathon for students to create innovative, sustainable, and most of all, useful applications or widgets for Cal students**. Student-developed app could be adopted by the University! Cash prizes start at $1,500.

Review of L4: Why Coroutines

Loop calls iterator function to get the next item

- eg the next token from the input

The iterator maintains state between two such calls

- eg pointer to the input stream.

Maintenance of that state may be difficult

- see the permutation generator in L4

See also Python justification for coroutines

- called generators in Python

Review of L4: Three coroutine constructs

co = coroutine(body)   lambda --> handle

creates a coroutine whose body is the argument lambda,

returns a handle to the coroutine, which is ready to run

yv = resume(co, rv)   handle x value --> value

resumes the execution into the coroutine co, passing

a to co's yield expression (v becomes return value of yield)

rv = yield(yv)   value --> value

transfers control to the coroutine that resumed into the current coroutine, passing yv to resume

Example

```
f(1,2)

def f(x,y) {
  cg=coroutine(g)
  resume(cg,x,y)
}
```

```
def g(x,y) {
  ck=coroutine(k)
  h(x,y)
  def h(a,b) {
    3 + yield(resume(ck,a,b))
} }
```

```
def k(x,y) {
   yield(x+y)
}
```

The call `f(1,2)` evaluates to 3.

Corner-case contest

Identify cases for which we haven't defined behavior:

1) yield executed in the main program

   let's define main as a coroutine; yield at the top level thus behaves as exit()

2) resuming to itself*

   illegal; can only resume to coroutines waiting in yield statements or at the beginning of their body

3) return statement

   the (implicit) return statement yields back to resumer and terminates the coroutine

*exercise: test your PA2 on such a program

States of a coroutine

How many states do we want to distinguish?

suspended, running, terminated

Why do we want to distinguish them?

to perform error checking at runtime:

- do not resume into a running coroutine
- do not resume into a terminated coroutine

The timeline

```
cg=coroutine(g)

v=resume(cg,1,2)







resume continues
print
```

```
def g(x,y) {

coroutine(k)

resume(ck,a,b)

resume continues

yield

}
```

```
def k(x,y){

yield(x+y)

}
```

Are coroutines like calls?

Which of resume, yield behaves like a call?

`resume`, for two reasons:

- like in regular call, control is guaranteed to return to `resume` (unless the program runs into an infinite loop)
- we can specify the target of the call (via corou. handle)

`yield` is more like return:

- no guarantee that the coroutine will be resumed (eg, when a loop decides it need not iterate further)
- yield cannot control where it returns (always returns to its resumer's `resume` expression)

Language design question (1)

Since resume behaves like a call, do we need to introduce a separate construct to resume into a coroutine? Could we just do this?

```
co = coroutine(…)     // creates a coroutine
co(arg)               // resume to the coroutine
```

Yes, we could. But we will keep `resume` for clarity.

Compare: in Python generators, resume is a method call .next() in a generator object. Do you like the fact that resume appears to be a call?

Language design question (2)

In 164/Lua, a coroutine is created with corutine(lam).

In Python, a coroutine is created by a call to function that syntactically contains a yield:

```
def fib():          # function fib is a generator/corou
    a, b = 0, 1
    while 1:
        yield b           # … because it contains a yield
        a, b = b, a+b

it = fib()           # create a coroutine
print it.next();     # resume to it with .next()
print it.next(); print it.next()
```

Language design question (2, cont'd)

Python creates coroutine by calling a function with yield? How does it impact programming?

What if the function with yield needs to call itself recursively, as in permgen from Lecture 4?

```
def permgen(a,n) { ... yield(a) ... permgen(a,n-1) ... }
def permutations(a) {
    def co = coroutine( permgen )
    lambda () { resume(co, a) }
}
```

You should know a workaround from this limitation

That is, how would you write permgen in Python?

See how Python writes recursive tree generators

Exercise on Lua vs. Python

A recursive Lua Fibonacci iterator:

```
def fib(a,b) {
     yield b
     fib(b,a+b)
}
def fibIterator() {
    def co = coroutine( fib )
    lambda () { resume(co, 0, 1) }
}
```

Rewrite it into a recursive Python generator.

Symmetric vs. asymmetric coroutines

Symmetric: one construct (yield)

- yield(co,v): can yield to an arbitrary coroutine
- all transfers happen with yield (no need for resume)

Asymmetric:

- resume transfers from resumer (master) to corou (slave)
- yield(v) always returns to its resumer

A language sufficient to explain coroutines

Language with functions of two arguments:

```
P ::= D*, E          sequence of declarations followed by an expression
D ::= def f(ID,ID) { P }

E ::= n
    | ID(E,E)
    | def ID = coroutine(ID)
    | resume(ID,E,E)
    | yield(E)
```

Implementation

- historical perspective on Lua and stackless Python

- Stackless Python is a controversial rethinking of the Python core (PEP 255)

High-level outline of coroutine interpreter

Each coroutine has its own instance of interpreter

  each instance evaluates an AST with bottom-up traversal

Interpreter instances share the environment

  scoping and frames work as in PA1

Each instance has its own call stack

  a coroutine can make calls, even recursive ones

resume(co) calls co's interpeter, which returns at yield

  the idea is to somehow resume co's interpreter

Simple, elegant. Almost works. Where's the problem?

  let's analyze where this implementation breaks down …

An attempt for a recursive corou. interpreter

```
def eval(node) {
    switch (node.op) {
      case ‘n’:    node.val
      case ‘call’: def t1 = eval(node.arg1)
                   def t2 = eval(node.arg2)
                   call(lookup(node.fun), t1, t2)
     case ‘resume’: co = eval(n.arg).code    get handle (this is OK)
                          eval(co)  resume (works only on first resume to co)
     case ‘yield’: return eval(node.arg)  returns to eval in same coro,
                                          not to resumer
   }
   def call(fun, v1, v2) { … eval(…) … }
}
```

Draw env and call stacks at this point

```
f(1,2)
def f(x,y) {
  cg=coroutine(g)
  resume(cg,1,2)
}
```

```
def g(x,y) {
  ck=coroutine(k)
  h(x,y)
  def h(a,b) {
    yield(m(a,b)+resume(ck,a,b))
} }
```

```
def k(x,y) {
   yield(x+y)
}
```

Draw the environment frames and the calls stacks.
The Python call stacks contain the recursive eval() calls, including the variables t1 and t2 (see the prev slide).
The recursive calls of eval() define the interpreter context.

The <u>recursive</u> PA1 interpreter

Program represented as an AST

– evaluated by walking the AST bottom-up

Where is the state of the interpreter maintained?

- on the call stack (created when Python eval() calls itself)

What's in the state?

1) temporary values, eg t1, t2 two slides ago
2) knowledge of which AST nodes are being evaluated
3) the call stack of the program being *interpreted*

Why PA1 fails

The interpreter for a coroutine c must return

- when c encounters a yield

The interpreter will be called again

- when some coroutine invokes resume(c, …)

Problem:

returning from the (deep) evaluation recursion would lose the interpreter state that is kept in the Python call stack

(we need to keep the eval(), eval(), eval() context somewhere)

Changes to interpreter

We must not rely on the Python call stack

so that we can return from a coroutine's interpreter while keeping its state stored somewhere, to resume later

Note that explicit stack is ok

we can preserve such a data structure across return-resume

Decisions, decisions:

1) temp values will go to "temporary registers" (bytecode)
2) the position of the evaluation will be kept in a PC
3) the call stack will be in an explicit stack (it will store return addresses)

AST vs. bytecode

Abstract Syntax Tree:

values of sub-expressions do not have explicit names

Bytecode:

- list of instructions of the form x = y + z
- x, y, z can be temporary variables invented by compiler
- temporaries stores values that would be in the stack of the stack-based interpreters

Example: AST `(x+z)*y` translates to:

```
$1 = x+z   // $1, $2: temp vars
$2 = $1+z  // return value of AST is in $2
```

Compile AST to bytecode

Traverse AST, emit code rather than evaluate AST

- when the generated code is executed, it evaluates the AST

What's the type of translation function **b**?

- b :: AST -> (Code, RegName)

The function produces code c and register name r s.t.:

- when c is executed, the value of tree resides in register r

Compile AST to bytecode (cont)

```
def b(tree):
      switch tree.op:
      +:    (c1,r1) = b(t.left)
            (c2,r2) = b(t.right)
            r = freshReg()
            instr = “%s = %s1 + %s\n” % (r, r1, r2)
            return (c1 + c2 + ist, r)
      n:    complete this so that you can correctly compile 2+3+4


      =:     complete this so that you can correctly compile
            x = 1+3 +4
            y = x + 2
```

Bytecode interpreter

What changes shall we make to the recursive one?

Must maintain:

- program counter
- explicit call stack

The rest pretty much the same as in PA1

Reading

Required:

see the reading for Lecture 4

Recommended:

Implementation of Python generators

Fun:

continuations via continuation passing style

Summary

Implement Asymmetric Coroutines

– why a recursive interpreter with implicit stack won't do

Compiling AST to bytecode

- btw, compilation to assembly is pretty much the same

Bytecode Interpreter

- can exit without losing its state