CSC326 Generators and Coroutines

Jianwen Zhu <jzhu@eecg.toronto.edu>

Revision History

Revision 1.1	2017-10	JZ

Table of Contents

1. Agenda

- Overview
- Generators, Coroutines and Pipeline
- Coroutine in C
- MultiTask Operating System
- Concurrent Nonblocking Programming with Coruoutine
- Probabilistic Programming Revisited

2. Overview

Motivating Problem: C10K Problem

- If your web service company is successful, need to scale
  - 10K simultaneous connection per server
  - 1M targeted for today
- Apache Web Server
  - Request Spawn a Process / OS Thread
  - Heavy Weight: Memory Overhead
  - Context Switch Overhead
- NginX Web Server
  - Single-Thread, Event-Driven
  - What happens if handler needs to block for IO Access?
  - Either entire thread block
  - Or need to remember state to come back
  - Fast but hard to program
- Ultra Light Weight MultiTask OS
  - Multi-Tasking: Illusion of multiple independent tasks
  - Context Switch: Traps/Interrupts
- What If we develop a MultiTask OS
  - Each request create its own task
  - Task traps for IO, at which point context-switch to other tasks
  - Scales to tens of thousands

3. Generators, Coroutines and Pipeline

Read David Beazley "A Curious Courses of Coroutines and Concurrency", Part 1/2 (Page 15-41).

4. Coroutines in C

Now that we see coroutine is powerful. Let’s dive inside and see how it might be implemented. We will consider the implementation of coroutine in C.

Consider a pipeline composed of a decompressor and a parser (strictly speaking, it should be called a lexer).

The decompressor takes a stream of characters as input and produces a stream of charactors as output. In processing each input character, it identifies a special character 0xFF, after and expanding the following two characters (len, c) by repeating c by len times.

The parser takes an input stream of characters, and produces a stream of tokens (WORD or PUNCT) as output.

Naturally, the algorithm of each pipeline stage can be expressed as the following pseudo code.

4.1. Parser/Compressor example

```
void decompress( void ) {
    /* Decompression code */
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                emit(c);
        } else
            emit(c);
    }
    emit(EOF);
}
```

```
void parse( void ) {
    /* Parser code */
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
                add_to_token(c);
                c = getchar();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
    }
}
```

4.2. Rewriting

Apparently, even though naturally expressed, the two function cannot work together using conventional function calls: * replacing emit() call in decompress by parse() call * replacing getchar() call in parse by decompress() call.

The conventional answer is to rewrite one of the ends of the communication channel so that it’s a function that can be called.

```
int decompress(void) {
    static int repchar;
```

```
    static int replen;
    if (replen > 0) {
        replen--;
        return repchar;
    }
    c = getchar();
    if (c == EOF)
        return EOF;
    if (c == 0xFF) {
        replen = getchar();
        repchar = getchar();
        replen--;
        return repchar;
    } else
        return c;
}
void parse(int c) {
    static enum {
        START, IN_WORD
    } state;
    switch (state) {
        case IN_WORD:
        if (isalpha(c)) {
            add_to_token(c);
            return;
        }
        got_token(WORD);
        state = START;
        /* fall through */

        case START:
        add_to_token(c);
        if (isalpha(c))
            state = IN_WORD;
        else
            got_token(PUNCT);
        break;
    }
}
```

You only need to modify one of them. Examing each of the modification, we can see that the modified code allows the function to be called repeatedly, and each call produces exactly one output. In the meantime, the modified remembers the state (context) of the execution, such that each subsequent call starts at where it is left of.

They do work, however, the way they were written is non-intuitive and error prone. Ideally, we want them to be coded in a similar way as the pseudo code.

4.3. Coroutine

Now let’s take a detour by looking at a much simpler example. Consider the following simple example: a loop that produces one integer at a time:

```
int function(void) {
    int i;
    for (i = 0; i < 10; i++)
        return i;   /* won't work, but wouldn't it be nice */
}
```

If this function is called repeatedly, what does it return?

Always 0, as each time function is called, i, as an "auto" local variable, is always re-initialized to be 0, and it aways starts executing from the beginning. In other words, the "context" of executing the function was fogotten!

4.4. First Attempt

Now let’s enhanced it, by * using static storage class for integer i, so that i keeps value next time the function is called; * introduce a state variable remembering where the program point should be next time function is called.

```
int function(void) {
    static int i, state = 0;
    switch (state) {
        case 0: goto LABEL0;
        case 1: goto LABEL1;
    }
    LABEL0: /* start of function */
    for (i = 0; i < 10; i++) {
        state = 1; /* so we will come back to LABEL1 */
        return i;
        LABEL1:; /* resume control straight after the return */
    }
}
```

4.5. Duff’s Device

The above code works, but looks ugly.

Stare at the following source code, what is strange? Do you think it will ever compile?

```
    switch (count % 8) {
        case 0:        do {  *to = *from++;
        case 7:              *to = *from++;
        case 6:              *to = *from++;
        case 5:              *to = *from++;
        case 4:              *to = *from++;
        case 3:              *to = *from++;
        case 2:              *to = *from++;
        case 1:              *to = *from++;
                    } while ((count -= 8) > 0);
    }
```

The switch statement is interleaved with the do-while loop. But it actually is perfectly legal C program!

The case statement is best considered as just labels that can placed anywhere in the code. This programming "device", is named after its inventor.

4.6. Second Attempt

Let’s see whether we can leverage Duff’s device to enhance our first attempt.

```
int function(void) {
    static int i, state = 0;
```

```
    switch (state) {
        case 0: /* start of function */
        for (i = 0; i < 10; i++) {
            state = 1; /* so we will come back to "case 1" */
            return i;
            case 1:; /* resume control straight after the return */
        }
    }
}
```

How we have "interleaved" the switch statement, together with the normal C code. Note "case 1:" is embedded within the for loop, which is in a different hierarchy from the "case 0:". But it is legal.

4.7. Third Attempt

Now let's use the C macros so that the code looks cleaner.

```
#define crBegin static int state=0; switch(state) { case 0:
#define crYield(i,x) do { state=i; return x; case i:; } while (0)
#define crFinish }
int function(void) {
    static int i;
    crBegin;
    for (i = 0; i < 10; i++)
        crYield(1, i);
    crFinish;
}
```

4.8. Final Result

It is really tedious to number the states corresponding to the different program locations.

Let's leverage the built-in C-Preprocessor macro. *LINE* corresponding to the line number of the source at time of macro expansion.

```
#define crBegin static int state=0; switch(state) { case 0:
#define crYield(x) do { state=__LINE__; return x; \
                        case __LINE__:; } while (0)
#define crFinish }

int function(void) {
    static int i;
    crBegin;
    for (i = 0; i < 10; i++)
        crYield(i);
    crFinish;
}
```

Now the code almost identical to the original C code.

4.9. Back to Pipeline Example

Now we can rewrite the decompress() function or the parse() function, as follows.

```
int decompress(void) {
    static int c, len;
    crBegin;
```

```
    while (1) {
        c = getchar();
        if (c == EOF)
            break;
        if (c == 0xFF) {
            len = getchar();
            c = getchar();
            while (len--)
                crReturn(c);
        } else
            crReturn(c);
    }
    crReturn(EOF);
    crFinish;
}

void parse(int c) {
    crBegin;
    while (1) {
        /* first char already in c */
        if (c == EOF)
            break;
        if (isalpha(c)) {
            do {
                add_to_token(c);
                crReturn();
            } while (isalpha(c));
            got_token(WORD);
        }
        add_to_token(c);
        got_token(PUNCT);
        crReturn( );
    }
    crFinish;
}
```

5. MultiTask Operating System

Read David Beazley "A Curious Courses of Coroutines and Concurrency", Part 5/6/7 (Page 92-168).

6. Probablistic Programming Revisited

It is often needed to develop a "model" of the world using probability to account for inherent uncertainty, imperfect knowledge, inaccuracies in input data, or the abstraction over insignificant details. We have seen an example of doing so in the "Continuation Passing Style" lecture, where we could find the distribution of complex random process by an exhaustive execution of a probabilistic program.

In this lecture, we want to see if we can accomplish the same using coroutine.

6.1. Lawn Model Example

Consider the modeling of a front lawn, whose grass may be wet because it rained or because the sprinkler was on. Since our knowledge is not perfect, we allow for 10% chance that the grass may be wet for some other reason. Also, if the rain was light, the grass might have dried up by the time we

observe it; a sprinkler may too leave the grass dry if, say, water pressure was low. We admit these possibilities, assigning them probabilities 10% and 20%, respectively. Suppose we observe the grass is wet. What are the chances it rained?

6.2. The Lawn Model in C

We can write the same lawn model as a C program:

```
bool grass_model (void) {
    bool rain      = flip(0.3);
    bool sprinkler = flip(0.5);
    bool grass_is_wet =
        (flip(0.9) && rain) || (flip(0.8) && sprinkler) || flip(0.1);
    if( !grass_is_wet )
        fail();
    return rain;
}
```

6.3. Inference

In the model, rain and sprinkler are independent boolean random variables, with discrete prior distributions [(0.3, true); (0.7, false)] for rain and [(0.5, true); (0.5, false)] for sprinkler. The variable grass_is_wet is the dependent random variable. The conditional statement asserts the evidence, using fail () to report the contradiction of model's prediction with the observation.

While the forward execution of the above grass_model() model is simple, it only produces one sample of grass_is_wet. On the other hand, what we really want is given grass_is_wet is true, what is the conditional probability of rain is true. In other works, we need the answer backwards!

One way of doing is using Monto Carlo simulation. If we could find a way to execute grass_model() is such a way so that ALL cases of (all executing paths — sound familiar?) the input boolean random variables that leads to grass_is_wet is true is enumerated, we just need to tally them up and count how many times rain is true.

Take the action as below:

```
void runit(void) {
    bool result = grass_model();
    printf("%g %c\n",MyWeight,result ? 'T' : 'F');
}
```

The standard output of the program is a sequence of lines consisting of two fields separated by a space. The first field is the probability, the second field is the outcome, the letter T or F. The output needs to be post-processed and collated — for example, by piping the output to the following AWK program. In the spirit of UNIX, each program ought to do one thing: generate results or post-process them.

```
awk '{r[$2] += $1}; END {printf "T: %g; F: %g",r["T"],r["F"]}'
```

The produced output is: T: 0.2838; F: 0.322

6.4. Enumerating All Execution Path

The magic of exploring all program execution path is to use the fork() system call.

```
static double MyWeight = 1.0;   /* changed at the start-up of each process */

bool flip(const double p) {
  pid_t pid = fork();
  if(pid == 0) {                /* in the child process, to handle False */
    MyWeight *= 1.0 - p;
    return false;
  }
  else if(pid == (pid_t)(-1)) {
    perror("fork failed");
    abort();
  } else {                      /* in the parent process, to handle True */
    MyWeight *= p;
    return true;
  }
}

int main(void) {
  int status;
  runit();
  while (wait(&status) > 0)
    ;
  return 0;
}
```

6.5. Probabilistic Inference with Coroutine

The magical fork() is nice, but expensive. The overhead of cloning an operating system process is very large and it does not take long that the operating system resource is exhausted for the probabilistic inference problem.

Now recall we have implemented ultra lightweight process using coroutine. How can we leverage it to resolve the probabilistic inference problem?

This would be a challenge I left to you at the end of the entire course:

- Solve the lawn model problem in Python using coroutine;
- Solve the lawn model problem in Python using continuation passing style.

Whether you can solve the challenge independently is a good benchmark for your performance of the course.

7. Disclaimer

The above lecture notes has used selected materials originally published by the following authors hereby acknowledged:

- David Beazley (http://www.dabeaz.com)
- Oleg Kiselyov (http://okmij.org/ftp)
- Simon Tatham (https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html)