George Beuhler
Scribe
1/12/12
CS 390

Lecture 2: Coroutines

Why should we be concerned about concurrency and parallelism. Fundamentally, a concurrent program provides no advantages over a sequential one. Any multi tape Turing Machine computation can be simulated using a single tape machine with only a quadratic increase in computation time. This shows one small benefit: parallelism is faster. A parallel system can be more responsive. If a web server receives two requests at the same time, a parallel implementation can process them both simultaneously and deliver content sooner than a sequential one. This example and its reduced mean service time is in lecture one. Also, parallel systems can be more flexible. A sequential data structure, such as a list, is less flexible than a parallel one such as a tree or a graph. The most important aspect of concurrency, however, is the power of expression. Concurrency gives us a more natural way to think about problems, which allows us to express them in an easier way. This allows for easier implementation.

We can start at our unit of control, a generator. A basic generator that increments by one is a trivial example. Suppose we want a more advanced situation: Generate all the elements in a tree. We need some kind of foreach loop that will generate the next element according to some predefined traversal policy, remember state, and keep things modular, abstract, and encapsulated. These generators act as crude threads. They have limited freedom, and they hold state and do a computation, and then wait.

Iterators are a special case of generators that allow enumerations of the elements of a data type. A simple example in C++ would be “a++”. They can be bidirectional, so “a--” is another example. How do we make these iterators remember state? How do we make a generalized version that works for all data types?

Procedures remember state using global variables and have a single operation: call. They use a single stack, and the stack frame is popped upon return. A generator has two operations: suspend and resume. They are asymmetric, which means they only work in one direction. The generator suspends itself, and then the caller resumes it. The generator cannot resume itself and the caller cannot suspend it. They use a single stack to remember state.

A coroutine is a glorified generator. It has one operation: transfer to another coroutine. They act like a generator as the coroutine saves its state, suspends, and then operations resume on a new one. This is what happens during a transfer. This operation is fully symmetric and can happen in both directions.

In order to implement coroutines, start with procedures. Procedures push and pop activation frames on the stack, which hold arguments and local variables. A frame is popped when the procedure returns. However, procedures are tied to sequential execution. To break the mold we use “remembering”, which is grabbing and restoring the stack. Now that we have this, we must figure out a way to preserve state that will be used on the next call. The answer is to use multiple stacks, one for each coroutine. This is the basic groundwork for creating threads.

Duff's Device is a famous C block of code that uses an unrolled while loop to efficiently process items in multiples of 8. Here is the device:

```
    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);
```

This can be used to build coroutines. We can use the switch statement to perform the jump to result in this primitive coroutine

```
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 */
        }
    }
}
```

(http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html)

This can be further cleaned up by using some macros to result in:

```
#define crBegin static int state=0; switch(state) { case 0:

#define crReturn(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++)
        crReturn(1, i);
    crFinish;
}
```

This coroutine remembers state and can use it to transfer control after the return.

A continuation is a reified representation of a programs control stack. We can define a primitive operation called call/cc(call with current continuation). Call/cc (fn k => e) will capture the current continuation, bind it to k, and evaluate e. fn k => e defines an anonymous function that takes k as an argument. We also have the operation throw k x. This applies continuation k with argument x.

Here is a continuation example with call/cc

call/cc (fn k => (throw k 3)+2)+1

Surprisingly, this returns 4. The computation is frozen at “+1” and is supplied the continuation k with argument 3. The “+2” is effectively discarded. Another way to think about this is time traveling to the point “add one to something”. Something, which is the continuation, is substituted with 3 to produce the final result of 4. Do not think of this example sequentially or it will never make sense.

Here is another example:

```
let f = call/cc (fn k => fn x => throw k (fn y => x + y))
in f 6 =>  12
```

Using primitives that grab and restore state, we can have procedures and manipulate threads. The trick is saving and restoring stacks.

A more advanced continuation of this material is found in the paper “Continuations and Concurrency” by Hieb and Dybig (1990). This paper delves into process continuations, which are more complicated continuation which remember the rest of a subcomputation. They are useful for tree-structure concurrency and better suited for the job than traditional continuations. They can use an alternative to call/cc called *spawn*. *Spawn* creates a new process subtree with a unique root. This technique is less powerful and more precise than call/cc, and still useful in sequential programming. For a traditional implementation, a stack of stacks called the “process stack” is used. Each stack in this stack is uniquely tied to the process controller. During an invocation, all stacks down to the target stack are removed and placed into a process continuation where its data is processed.

To implement this concurrently, use a process tree instead of stack. During an invocation, the subtree of stacks rooted at the labeled stack is pruned from the tree and packaged into a continuation.