Lecture 2: Coroutines
1/12/12
Expressivity

- A concurrent program provides no fundamental computability advantages over a sequential program
  - Any computation that can be expressed using a multitape Turing machine can be expressed using a single-tape Turing machine
- Why have it, then?
  - Performance: parallelism
  - Responsiveness:
    - web servers
    - operating systems
  - Flexibility
    - Algorithms, data structures, etc.
  - Expressivity
Starting Point: Control

• How do we represent or capture the notion of simultaneity?

• Example:
  – A generator:

  ```
  x = 0;
  proc f() = {
    r := x;
    x := x + 1;
    return r
  }
  
  f();  yields 0
  f();  yields 1
  ```
Generators

- Suppose instead of incrementing a counter, we returned the elements of an array

\[ i := 0 \]

\[
\text{proc } f(a) = \{ \text{ if } i < \text{Array.length}(a) \\
\quad \text{then } \{ r := A[i]; \\
\quad \quad i := i + 1; \\
\quad \quad \text{return } r \} \\
\quad \text{else raise ArrayOutofBoundsExn } \}
\]

- A bit more complicated, but generation can still be expressed using updates on the array index
Generators

• What happens if generation is not so apparent?

• Example:
  - Generate all the elements of a tree
  - How do we write a getNextTreeElt routine?
    • We need to record and remember the next position in the tree for subsequent calls to the generator
    • We want to keep things modular and abstract, and not expose how the tree is traversed to clients
    • Want to write something like:
      
      ```c
      foreach node in Tree do { ... }
      ```
Generators

- How do we implement foreach?
  - It is meant to be a generator that yields the next leaf in the tree (according to some traversal policy) every time it is invoked
  - Must preserve the state of the traversal internally
  - How should this state be kept?
Iterators and Coroutines

- Iterators are a special case of generators
  - Found in C++, Java, Python, Lua, etc.
  - Enables enumeration of the elements of a datatype

- Key questions:
  - How do we maintain local state implicitly?
  - Can we build a general iteration framework without having to provide specialized versions for each datatype?

- Idea:
  - Imagine a procedure that “remembers” its state across calls
Coroutines

A()                    B()               C()

what happens when A() calls B() again?

Procedure calls

Coroutines

A()                   B()                    C()

When a coroutine returns, it remembers its program state. Why is this useful?

CS390C: Principles of Concurrency and Parallelism
Generators and Coroutines

- Procedures:
  - single operation: call
  - single stack, stack frame popped upon return

- Generators:
  - two operations: suspend and resume
    - asymmetric: generator suspends, caller resumes it
  - single stack, generator is an “object” that maintains local state variables
  - single entry point

- Coroutines:
  - one operation: transfer
    - fully symmetric
  - When A transfers to B it acts like a:
    - generator suspend wrt A
    - generator resume wrt B
  - transfer names who gets control next
    - non stack-like
Coroutines and Concurrency

• How would you implement coroutines?
  − Typically, implementations of procedures and procedure calls involving pushing and popping “activation frames” on the stack
  − These frames hold the arguments and local variables for the call.
  − The frame is popped when the procedure is returned.

• How do we preserve the state that will be used when we make the next call?
  − Keep multiple stacks, one for each coroutine
  − Essential feature of threads
Continuations

• A reified representation of a program’s control stack.

• Example:

```plaintext
proc f(x) = { ... 
g(y); 
... ; A
}

proc h(y) = { ... 
f(...); 
... ; B
}
```

When g is called, the program stack retains enough information to “remember” that A must be executed and then B.

The stack captures the “rest of the computation” - it is the continuation of the call to g().

If the computation were preempted immediately after the call to g() returns, its resumption would entail execution of the continuation.
Can we reify this notion into a source language?

- result is a continuation, a reified representation (in the form of an abstraction) of a program control-stack.
- Define a primitive operation called call/cc:
  - call-with-current-continuation
  - callcc (fn k => e)
    - captures the current continuation, binds to k, and evaluate e
    - the notation fn k => e defines an anonymous function that takes k as an argument
  - throw k x
    - apply continuation k with argument x
Example

call/cc (fn k => (throw k 3) + 2) + 1 \rightarrow 4

let f = call/cc (fn k => fn x => throw k (fn y => x + y))
in f 6 \rightarrow 12
Example: Samefringe

- Two binary trees have the same fringe if they have exactly the same leaves reading left to right
Samefringe

- Simple strategy:
  - Collect leaves of both trees into two lists, and compare elements

  \[
  \text{frontier } t_1 = \text{collect all leaves of tree } t_1 \\
  \text{frontier } t_2 = \text{collect all leaves of tree } t_2 \\
  \text{samefringe}(t_1,t_2) = \text{compare the frontiers of } t_1 \text{ and } t_2 \text{ pairwise}
  \]

- What's wrong with this approach?
Samefringe Using Coroutines

- Rather than collecting all leaves or transforming tree eagerly, generate leaf values for two trees lazily

- Create generators for the two trees that yield the next leaf when invoked, and return control back to the caller, remembering where they are

```ml
fun samefringe(t1, t2) = 
  let val g1 = makeGenerator(t1)
  val g2 = makeGenerator(t2)

  fun loop() = let val l1 = g1()
                val l2 = g2()
                in case (l1,l2) of
                  (Empty,Empty) => true
                  | (Leaf(x),Leaf(y)) => (x = y) andalso loop()
                  | _ => false
                end

  in loop ()
  end
```

- How do we write these generators?

CS390C: Principles of Concurrency and Parallelism
What have we achieved?

- Call/cc gives us a way to capture the remaining part of a computation at any given program point.
- If we capture a continuation and store it, we have a handle on a program state. This is what is necessary to suspend a co-routine.
- If we invoke a captured continuation, we effectively resume a computation (or co-routine) at the point where its continuation was saved.
Generators (revisited)

- How should we generate the leaves for `samefringe`?
- Need to:
  - save state and suspend after each new leaf is found
  - resume caller once state is saved
- Use continuations for state saving
  - If continuations are “first-class” they can be stored into locations for later use
cont caller;
proc generate-num;

proc generate-numbers () = {
    loop (i=0) (i++)
    {
        call/cc (fn (genrest) =>
            { generate-num := proc () = throw genrest ();
                throw caller i }
        }
    }

generate-num := generate-numbers;
proc make-generator () = {
    fn () => call/cc (fn (k) => {
        caller := k;
        generate-num()
    })
}


CS390C: Principles of Concurrency and Parallelism
Analysis

- Continuations saved at two points in the generator definition:
  - To save state associated with the generator loop itself
  - To save the callers state, so that it can be resumed with the next generated value
- How might we implement this functionality in the absence of explicit continuations?
Manifestation

- Several languages provide explicit support for continuation-capture
  
  - Scheme, ML, Haskell, ...

- Here’s how we might write a generator in ML:

  ```ml
  fun makeGenerator i = 
    let val caller = ref NONE 
      val generateNumRef = ref (fn () => ()) 
      fun generateNums () = 
        let fun loop (i) = 
          let val _ = callcc (fn genrest => 
            (generateNumRef := (fn () => throw genrest ())); 
            throw (valOf(!caller)) (SOME i)) 
          in loop (i+1) 
            end 
        in loop (i) 
          end 
      val _ = (generateNumRef := generateNums) 
    in fn () => 
      callcc (fn k => (caller := SOME k; 
        (!generateNumRef)(); 
        NONE)) 
    end
  ```
What next?

• Being able to save and restore control state is at the heart of any implementation of concurrency

• Using continuations, we can build our own threading system
  - schedulers
    • cooperative
    • preemptive

• Not quite so simple though ...
  - exceptions and aborting; interrupts
  - asynchrony
Further Reading

Essentials of Programming Languages, Friedman, Wand, Haynes (2001)

Continuation-based Multiprocessing, Wand (1980)

Continuations and Threads: Expressing Machine Concurrency Directly in Advanced Languages (1997)

Continuations and Concurrency, Hieb and Dybvig (1990)