

# Coroutines

Lecture 3  
CS 390C

1/14/08

# Observations

- Last time, we claimed that concurrent programs provide no fundamental advantage over sequential ones.
- Why have it?
  - Performance: parallelism
  - Responsiveness
    - web servers
    - operating systems
  - Flexibility and expressiveness
    - Algorithms
    - Data structures
    - Control structures

# Background

- To understand why we should be interested in concurrency, we need to first understand what we can do with sequential programs
- Start with a simple example:
  - A generator:

```
val x = ref 0
fun f () = let val r = !x
            in x := !x + 1
            end
```

```
f() ;; yields 0
f() ;; yields 1
```

- Every time `f()` is invoked, `x` is updated

# Generators

- Suppose instead of incrementing a count, we returned elements of an array:

```
val i = ref 0
```

```
fun f(a) = if i < Array.length(a)  
          then let val r = Array.sub(a,i);  
                in (i := !i + 1; r)  
          end  
          else throw ArrayGenExn
```

- Things are a bit more complicated, but generation still can be expressed using just updates on the array index

# Generators (cont)

What happens if the generation of elements is not so apparent?

- Suppose we want to generate all elements of a tree?

```
datatype 'a tree = Empty | Leaf of 'a | Node of 'a tree * 'a tree
```

```
fun f(t: tree) = let val i = getNextElt(t)  
                in ....
```

- How do we write getNextElt?
  - We need to somehow record the next position in the tree
  - Want to preserve modularity and abstraction and not expose how the tree is traversed

# Generators

- Would like to write something like:

```
fun f(t:tree) = foreach n in t do
```

....

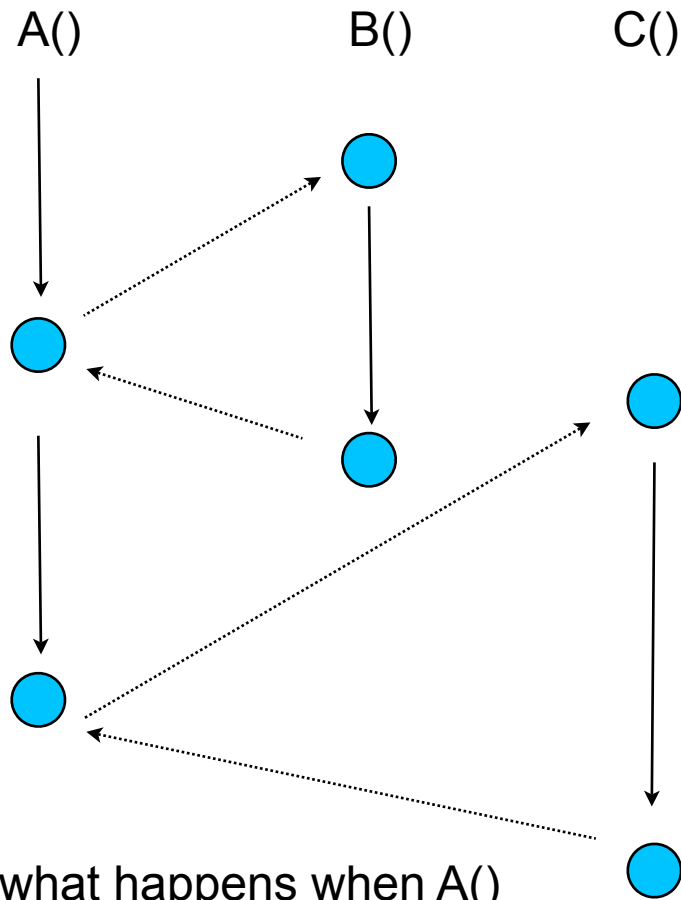
- But, how do we implement foreach?
  - it is a generator that yields the next element in the tree each time it is invoked.
  - must preserve state to yield the tree
    - where should this state be kept?

# Iterators and Coroutines

- Iterators are a special case of generators
  - originally proposed in CLU
  - found in C++, Java, Python, Lua, etc.
- Key questions:
  - How do we maintain local state implicitly?
- One possible solution:
  - think of iterator as a special kind of procedure that “remembers” its state across calls
  - how do we do this modularly?

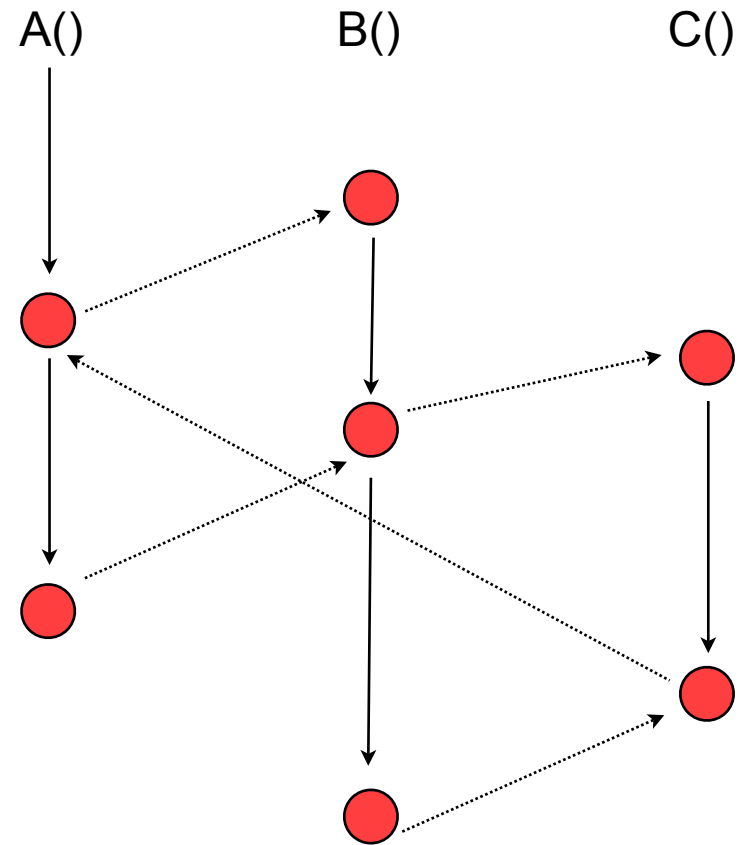
# Basic Idea

Procedure calls



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

Coroutines



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

# 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*



# Samefringe

- First approach:
  - Collect leaves of both trees into two lists, and compare elements

```
datatype tree = Empty | Leaf of int | Node of tree * tree
```

```
fun frontier(Empty) = []  
  | frontier(Leaf(x)) = [x]  
  | frontier(Node(t1,t2)) = append(frontier(t1),frontier(t2))
```

```
fun samefringe(t1,t2) = let val l1 = frontier(t1)  
                          val l2 = frontier(t2)  
                          fun compare(hd1::tl1,hd2::tl2) =  
                            (case hd1 of  
                             Empty => (hd2 = Empty)  
                             | Leaf(x) => (case hd2 of  
                                           Leaf(y) => (x = y)  
                                           andalso compare(tl1,tl2))  
                            in compare(l1,l2)  
                          end
```

- What's wrong with this approach?

# Samefringe (cont)

```
fun samefringe(Empty,Empty) = true
  | samefringe(Leaf(x), Leaf(y)) = (x = y)
  | samefringe(Node(Empty,t1),Node(Empty(t2))) =
    samefringe(t1,t2)
  | samefringe(Node(Leaf(x),t1),Node(Leaf(y),t2)) =
    (x = y) andalso samefringe(t1,t2)
  | samefringe(t1 as Node(_,_), t2 as Node(_,_) =
    samefringe(adjust(t1),adjust(t2))
  | samefringe(_,_) = false
```

and

```
fun adjust(x as Node(Empty,_)) = x
  | adjust(x as Node(Leaf _, _)) = x
  | adjust(Node(Node(t1,t2),t3)) = adjust(Node(t1,Node(t2,t3)))
```

What's wrong with this solution?

# 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

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

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

- How do we write these generators?

# Generators and Coroutines

- Procedures:
  - single operation: call
  - single stack, stack frame popped upon return
- Generators:
  - two operations: suspend and resume
    - assymetric: 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

# Continuations

- Revisit our original question:
  - What is unique about concurrency?
  - Can we express concurrent behavior using sequential primitives
    - How do we capture the notion of multiple control stacks?
    - Answer: continuations