Continuing Continuations...

Overview

- How to...
  - Build user-level threads
  - Write your own backtracking procedure

Threads

- Reminder
  - Continuation represents the future of the computation
- Threads
  - Several (pseudo) concurrent *agents*
  - Each *agent* has its own future
- Purpose of a thread library
  - Multiplex the various thread onto a physical agent
    - Physical agent: The actual OS thread we have
    - Logical agents = user threads = continuations

Ingredients

- What we need
  - A data structure to track all agents
    - What should we use ?
  - A data structure to represent the execution context of an agent
    - What should we use ?
  - Functions to
    - Create a new thread (agent)
    - Yield to another agent
    - Self-termination of threads
  - In a real world
    - Synchronization primitives (mutex / semaphore /...)

The Execution Context

- This is simply a continuation!
  - When should we capture it ?
  - What to do with it once it has been captured ?

Switching

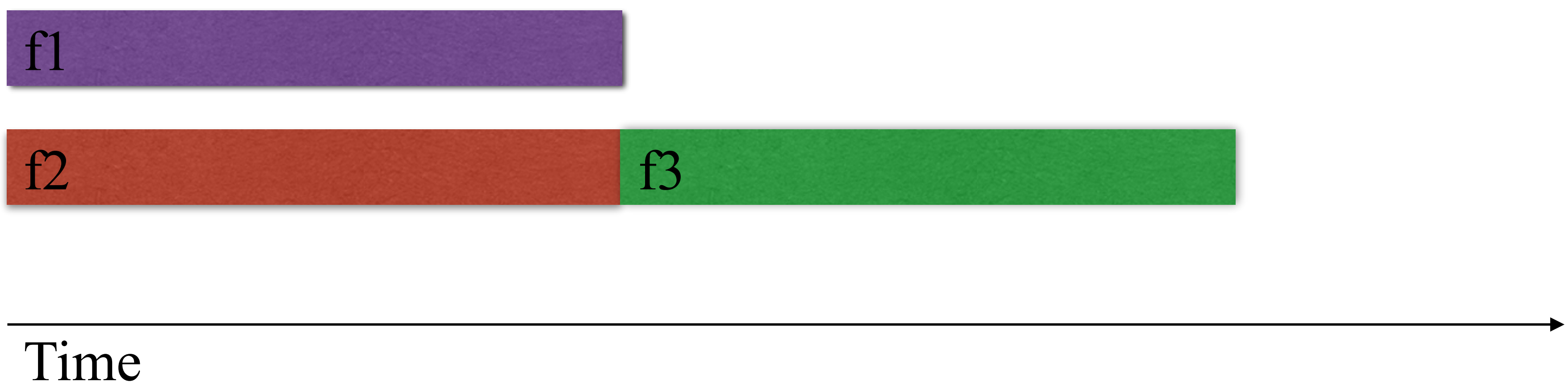
- What's up after saving the context ?
  - Switch to another thread!
  - How ?
    - Gets its context and restore it....
    - So, in terms of continuation ?

The Big Picture

```
(* Reference to a queue representing the ready threads *)
val Queue : bool Cont list ref = ref nil
fun thread_create f = (* Create a thread to execute f *)
fun thread_exit ()  = (* Terminates the current thread *)
fun thread_yield () = (* Voluntarily relinquish control to
                         another ready thread *)

fun loopPrint m n =
     if n<10
     then let val _ = print (m ^ (Int.toString n) ^ "\n")
          in  (thread_yield(); loopPrint m (n+1))
          end
     else ()

fun f1  () = loopPrint "hello f1:" 0
fun f2  () = loopPrint "hello f2:" 0
fun f3  () = loopPrint "hello f3:" 0

fun main() = let val _  = thread_create f1
                 in (f2();f3())
               end
```

Pictorially

- Two threads
  - Main one runs **f1**
  - Second thread runs **f2** followed by **f3**



The Big Picture

```
(* Reference to a queue representing the ready threads *)
val Queue : bool Cont list ref = ref nil
fun thread_create f = (* Create a thread to execute f *)
fun thread_exit ()  = (* Terminates the current thread *)
fun thread_yield () = (* Voluntarily relinquish control to
                         another ready thread *)

fun loopPrint m n =
     if n<10
     then let val _ = print (m ^ (Int.toString n) ^ "\n")
          in  (thread_yield(); loopPrint m (n+1))
          end
     else ()

fun f1 () = loopPrint "hello f1:" 0
fun f2 () = loopPrint "hello f2:" 0
fun f3 () = loopPrint "hello f3:" 0

fun main() = let val _ = thread_create f1
                 in (f2();f3())
                end
```

Desired Output

- Function f1 executes in a separate thread
  - F1 is concurrent with the sequence f2();f3()
  - f1,f2,f3 all print the first 10 natural numbers
  - f1/f2 strictly alternate
  - When f1 is done f2 is done as well.
  - After f1 terminates, the thread dies
  - f3 executes 'alone'.

Tracking Threads

- We need a run Queue!
  - Keep track of a queue of continuations

```
val callcc    = SMLofNJ.Cont.callcc
val throw     = SMLofNJ.Cont.throw
type 'a cont = 'a SMLofNJ.Cont.cont

val Queue : bool cont list ref = ref nil

fun enQueue p = Queue := (!Queue) @ [p]

fun deQueue () =   let  val (a::bs) = !Queue
                        val _        =  Queue := bs
                    in a
                   end
```

Creating a Thread

- What must be done
  - Add a new continuation to the queue for the new thread
  - Dispatch to either the new or current thread.

```
fun thread_create f =
    let val c = callcc ( fn k => (enQueue k; true) )
    in    if c
          then (f(); thread_exit())
          else ()
    end
```

Yielding

- Purpose
  - Voluntarily relinquish control for another thread
- Implementation ?
  - Schedule self back on queue
  - Get the next guy out of the queue
  - Transfer control to him

```
fun thread_yield () =
    let val c = callcc ( fn k => (enQueue k; true) )
    in    if c
          then throw (deQueue()) false
          else ()
    end
```

Exiting

- Nothing much to do...
  - If there is something else in the queue...
    - De-queue it and schedule it.
    - Otherwise go on sequentially.

```
fun thread_exit () = if null (!Queue)
                     then ()
                     else throw (deQueue ()) false
```

Overview

- How to...
  - Build user-level threads
  - Write your own backtracking procedure

Backtracking Search

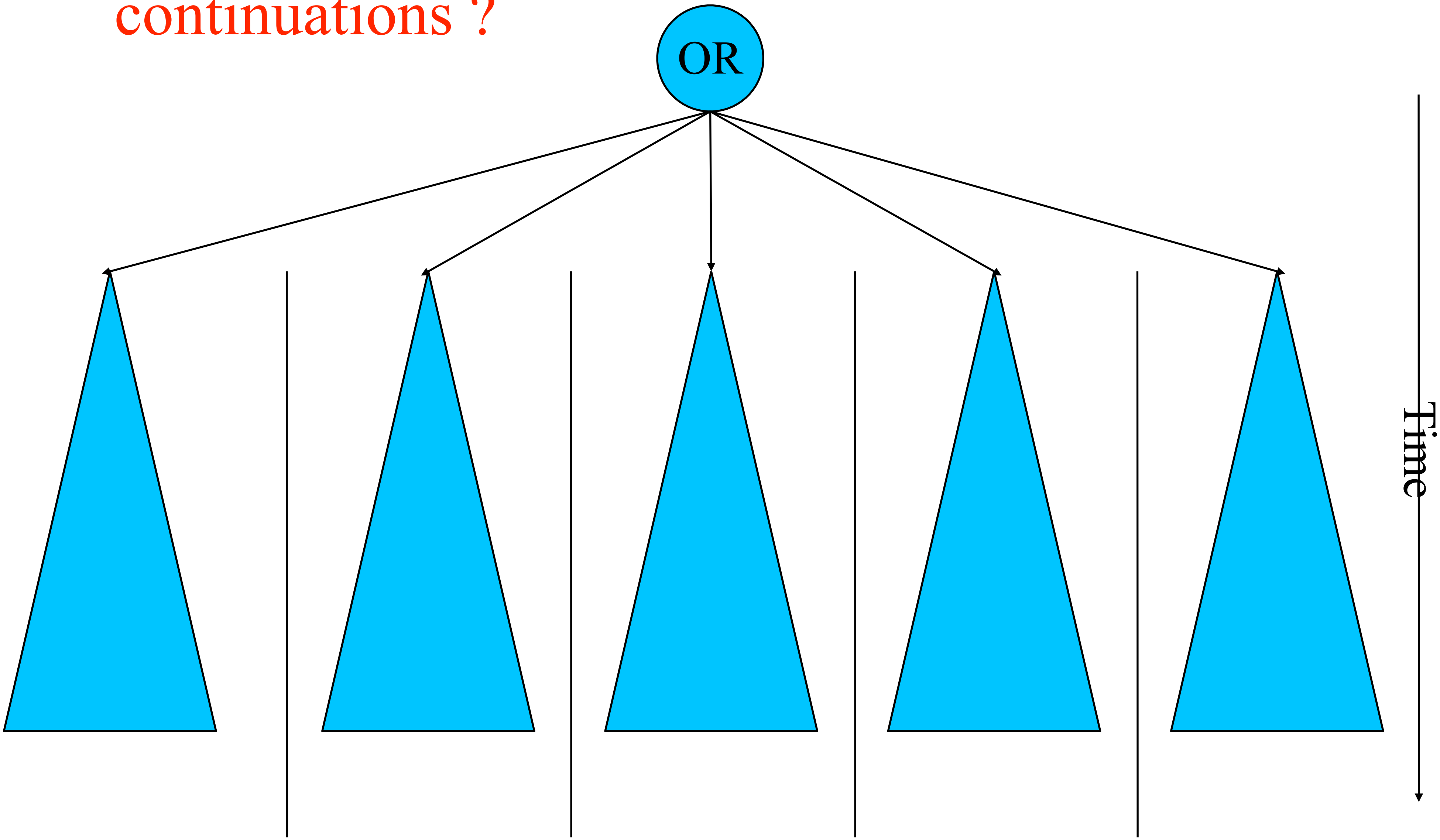
- Objective
  - Program the backtracking ourselves
- Pay-off
  - Backtracking search in any language with continuations
  - Essence of non-deterministic style available

Ingredients

- What we need
  - A labeling function to
    - Try all possible values for a given variable
  - A mechanism to backtrack to a choice
    - When backtracking: consider the remaining alternatives
    - When no choice points left, simply carry on.
  - Choice point accumulate
    - Going down the and-or tree add new choice points
    - In what order do we reconsider the choice point ?
      - So, what data structure is appropriate ?

Or-Nodes

- What is the relationship between or-nodes and continuations ?



General Solution

- When reaching an or-node
  - Capture the future
  - Save it somewhere
  - Pick a branch
  - Make the choice of the branch
  - Execute the future
- When backtracking
  - Retrieve the last choice point
  - Pick the next branch
  - Make the choice of that branch
  - Execute the future

Saving A Possible Future

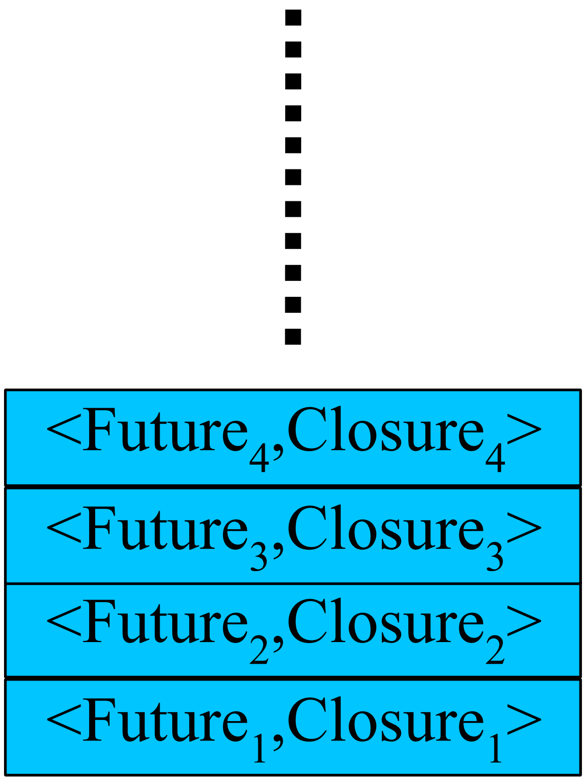
- We want to
  - Come back to last choice point
  - This is *chronological* backtracking
  - What is the right data structure to use ?

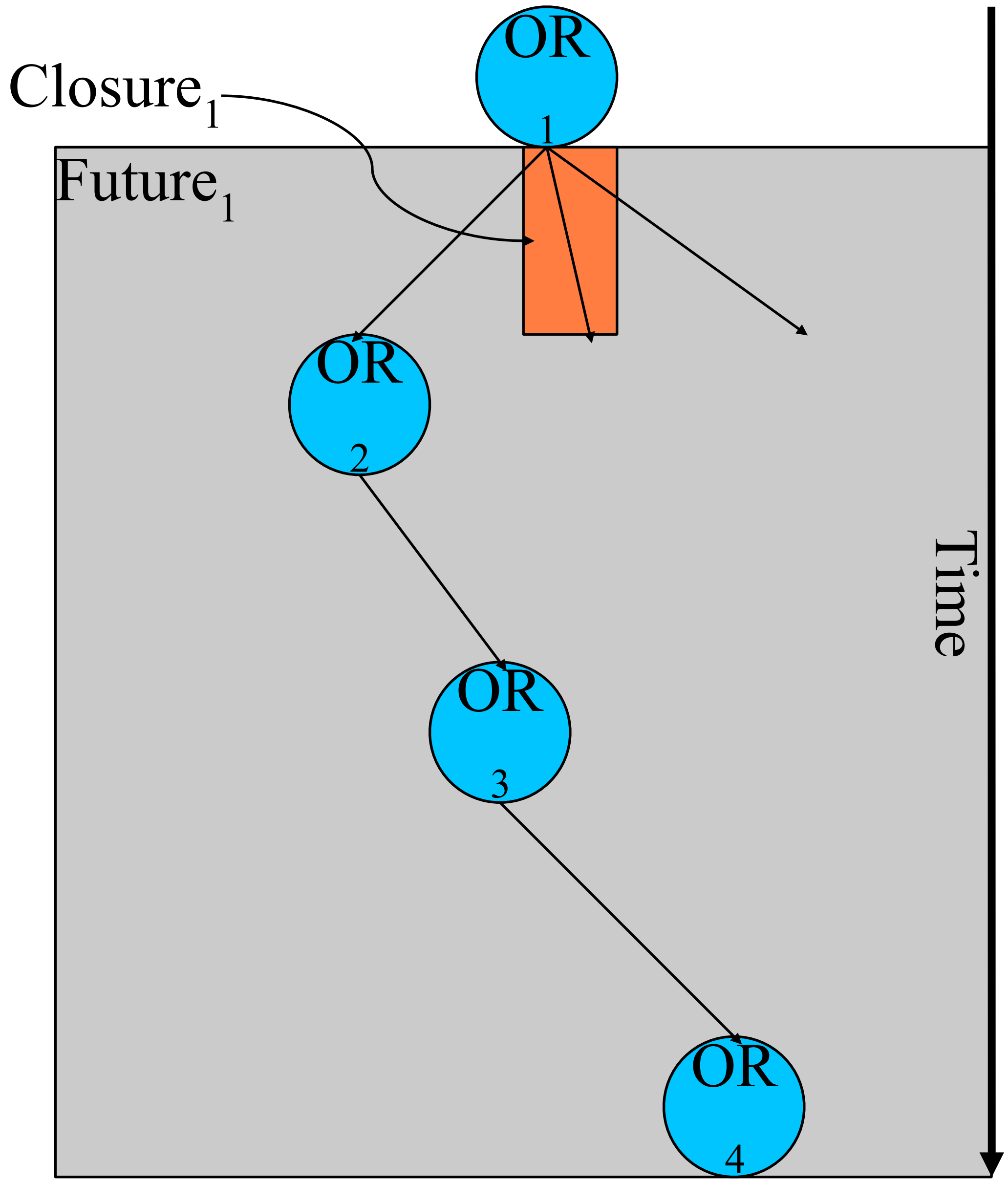
Picking Up the Next Branch

- How to capture the branch selection ?
  - A function to
    - Decide if more branches left after this one
    - Computing the value on the branch
    - If more branches left
      - Save the future again
  - What does this function need to have access to ?

CP Stack

- Pictorially





The ML Solution

- Ground zero
  - The Choice point stack

```
val callcc = SMLofNJ.Cont.callcc
val throw  = SMLofNJ.Cont.throw
type 'a cont  = 'a SMLofNJ.Cont.cont

type choice   = int cont * (int -> int) * int

val cpStack : choice list ref = ref nil

fun pushCP p = cpStack := p::(!cpStack)

fun popCP () = let val (a::bs)    = !cpStack
                   val _          = cpStack := bs
               in a
               end
```

Representing Variables

- Representing assignment as…
  - A simple binding
    - It will change upon backtrack
  - Store the "assignments" in a list
    - Convenient for labeling heuristics!
      - Variable ordering...
  - Set of values (Domain) ?
    - Could be stored in the variable (as a list of values)
    - Here we simply assign the value directly
      - consider D=[1..4]

```
let val vars = [(1,4),(1,4),(1,4),(1,4)]
in ...
end
```

Labeling a Variable

- The code...

```
fun label (lb,ub) =
    let val s = callcc
         (fn k => let fun branch c =
                                 if (c < ub)
                                  then (pushCP (k,branch,c+1);c)
                                  else ub
                            in (pushCP(k,branch,lb+1);lb)
                         end
         )
    in {value=s,dom=(lb,ub)}
    end
```

callcc returns
- The first value (first branch)
- The next value (later)

callcc saves
- the future + branch

Backtracking

- How to backtrack ?
  - Go back to the choice point stack
    - Pick up last choice
      - What if none left ?
    - Call the branch closure
    - Throw the value of the branch to the continuation

```
fun backtrack() =
     if null !cpStack
     then ()
     else let val (k,C,n) = popCP ()
          in throw k (C n)
          end
```

Finally: The Search

- Searching is
  - Labeling the variables in the list
  - Doing the testing / constraint propagation
  - Backtracking when we fail
- Example search
  - Simply label the list
  - Testing prints the list of variables (a tree leaf)
  - Backtrack to find the other leaves

Backtracking Search in ML

- Code

```
fun labelList [] = nil
  | labelList (a::bs) = (label a)::(labelList bs)

fun main () =
     let val l = [(1,4),(1,4),(1,4),(1,4)]
     in  let val x = labelList l
         in (print (printList x);backtrack ())
         end
     end

val _ = main()
```