Implementation issues in High Level Languages

Lecture 8

Key concepts this lecture

- Concurrency
- Evaluation order (eager vs lazy)
- Continuations
- Referential transparency
- Mutability
- Garbage Collection

Concurrency (1)

- Concurrency: executing different parts of a program in parallel
- Motivation: speed up execution
  - Multi-CPU supercomputers
  - Moore's Law => parallel CPUs. For last 10 years speedups come from increasing concurrency
  - Needed even on single PCs now with multi-core CPUs
- Motivation: improve interactive response
  - e.g. respond to GUI clicks while running long computation

Concurrency (2)

- When operations are executed concurrently order of execution cannot be fixed
- Sometimes result depends on order of execution

```
let mutable x = 0
let mutable y = 10
let f1() = x <- y+1
let f2() = y <- x+1
```

x=0; y=10

f1()	f2()
f2()	f1()
x=11 y=12	**x=2 y=1**

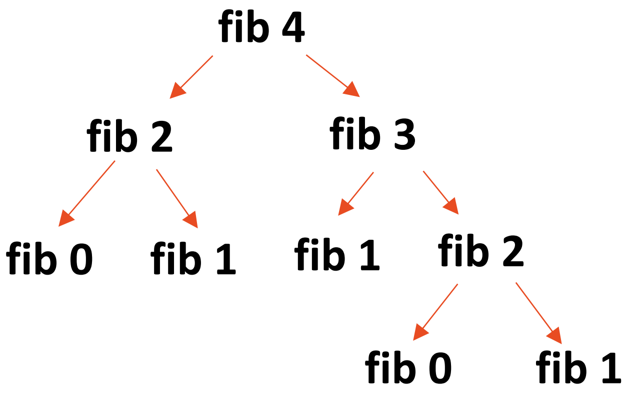
Concurrency (3)

- How can we write concurrent programs that work correctly?
  - 1. Complex answer: EIE Distributed Systems material
    - **Race conditions**
    - **Locks**
    - **Lots of complexity**
  - 2. Simple answer: use (pure) functional programming
    - Concurrent execution **always works** with deterministic result, without any special constructs

Evaluation order (1): Eager evaluation

- Eager evaluation: calculate expressions as soon as possible
- F# evaluation order (like most high level languages) is **eager**
  - Function parameters are evaluated before calling function (e.g. + here!)
  - + evaluation **must be eager** because the evaluated results are needed to execute +
    - + is therefore a **strict** operation
  - The order of evaluation of the two sides of + does not matter (and usually is not defined)
  - Therefore the two calls to fib here can be executed in parallel on different CPUs
- Note that control flow must still not evaluate unwanted branches
  - Here one of the two match cases is never evaluated, depending on n
  - n must be evaluated before the match
  - Otherwise function will not terminate

```
let fib n =
match n with
| 0 | 1 -> 1
| _ -> fib (n-1) + fib (n-2)
```



Evaluation order (2): Lazy evaluation

- Lazy evaluation
- Lazy means don't evaluate a function parameter unless it is absolutely needed to compute result
  - e.g. by a strict arithmetic operation itself known to be needed
- Why do we need lazy?
  - if then else is NOT strict
  - Compiler manages this inside a function
  - Compiler does not know how to do this if laziness crosses function boundary
  - e.g. does a long list need to be computed (because it is used)
  - Define infinite lists! (Favourite FP trick)

Why does this not work in F#?

```
let numbers n =
    n :: numbers n + 1
```

NB - in F# do this with seq!
or (better if results are reused) with LazyList

Evaluation order (3): Implementing lazy evaluation

- Lazy evaluation is implemented by passing a **continuation** function for each parameter.
  - Continuations are used to suspend parameter evaluation till it is needed
  - Calling the continuation is done when needed and this evaluates the parameter

```
let bExpCont = fun () -> bExp // continuation
// this works in F#
// implements laziness under programmer control
let ifFun (a: bool) (bCont: unit->T) (cCont: unit-> T) =
    if a then b'() else c'()
```

Here evaluation of b and c is not known until a is evaluated

Either b or c but not both is evaluated by calling the continuation b ' or c '

Full Lazy evaluation: compiler does this automatically and always without programmer needing to specify it or write continuations (Haskell)

Partial Lazy evaluation: programmer must specify this. Provide syntactic sugar to make it easier. Define lazy lists.

Evaluation order (4): F# support

1. DIY – use continuations and (see later) computation expressions
2. `Lazy('T)`

```
let x = 10
let result = lazy (x + 10)
printfn "%d" (result.Force())

// see here for more info
```

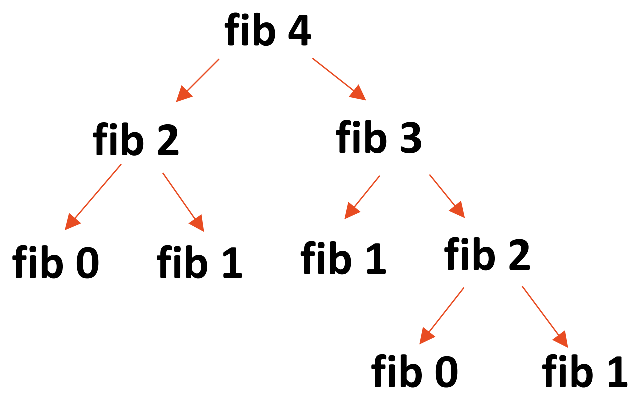
3. **`LazyList('T)` package**
    1. Results are cached to prevent repeated evaluation
    2. Better for efficiency
    3. May cause memory leaks
    4. See here
4. Seq module (**`seq<'T>`** type) Lists computed lazily **without cached results**

Referential transparency

improve efficiency by allowing data being copied or referenced as is convenient

- Replacing an **expression** by its **value** at any time cannot change the **result of the program**.
  - Referential transparency
    - Expression cannot contain (mutable) free variables
    - Expression cannot have significant side effects – e.g. change global variables on which program result depends
- Referential transparency means
  - If a program terminates, the result is independent of the evaluation order
  - Expressions can always be copied for execution on a new CPU with separate memory

```
let fib n =
match n with
| 0 | 1 -> 1
| _ -> fib (n-1) + fib (n-2)
```



What is mutability?

- Anything that requires the value of an expression to change is a mutation of the expression:
  - Changing the value of an array element
  - Changing the value of a writeable location in memory
- We will include **side-effects**: something significant which the evaluation of an expression causes to happen.

Why care about mutability?

- Why is **restricting mutability** a good idea?
  - Improve **evaluation order** independence
    - Much quoted "functional languages are good for concurrency"
  - Improve **referential transparency**
    - Improve efficiency by allowing data copying or reference as is convenient
  - Improve **polymorphic (generic) type inference**
    - Not often understood
    - The FSharp "value restriction" problem. In Java: contravariant vs covariant generics. See later.

Write-once assignment: when not all mutability is equal

- Use e.g. an array in which each element is written exactly once
- The (global) assignment here is in fact not such a problem
  - Referential transparency is not broken **as long as all writes are completed before the value is read**.
  - Given to this, the order of evaluation does not matter
  - Common application – Logic Programming (beyond scope of this module)
- We need a mechanism to generate an error if a value is written twice.

```
type 'T WriteOnce = | Empty | Value of 'T | Error
```

F# implementation

- This implementation allows **one write**, or **multiple same value** writes without error
- Write-once mutability
  - can be used in an eager parallel program
  - ensure the WriteOnce locations are not read until the whole program has executed
  - the result will be independent of execution order
  - note that "error" is a special value returned, not the program failing!

```
type 'T WriteOnce =
    | Empty
    | Value of 'T
    | Error

let WOLoc : int WriteOnce Ref = ref Empty

let woWrite loc x =
    let x' = !loc
    match x' with
    | Empty -> loc := Value x
    | Value y when x = y -> ()
    | _ -> loc := Error

let test() =
    let l0 = !WOLoc
    woWrite WOLoc 1
    let l1 = !WOLoc
    woWrite WOLoc 1
    let l2 = !WOLoc
    woWrite WOLoc 2
    let l3 = ! WOLoc
    printfn "%A" [l0 ; l1 ; l2 ; l3]
```

empty 1 1 Error

Garbage Collection

- **Heap memory** is used for variable sized objects like lists etc
- **What** is garbage collection?
  - Programmer never allocates or deallocates memory
  - Heap memory is automatically allocated
  - And automatically recycled when no longer used
- **Who** performs garbage collection?
  - Memory model is built into language
  - Memory model is implemented by **language run-time system**
- **Where** does garbage go?
  - blocks of memory are added to the heap "free memory"
    - Many various sized blocks
    - One Contiguous block – if compacting garbage collection

Already know about garbage collection? Have a look at RUST : a very clever High Level Systems Programming Language! More complex memory management strategies than any other language

Technology

- **Mark and scan**
  - Pauses execution for garbage collection
  - Low overhead
  - Overhead scales as size of: $\dfrac{U + \alpha F}{F}$
- **Copying**
  - Automatic compaction with copying
  - Pauses execution for garbage collection
  - Medium overhead
  - Overhead scales as size of: $\dfrac{U}{F}$
- **Reference count**
  - No pause required for garbage collection
  - High overhead (hardware support?)
  - Overhead independent of memory size
  - Complex

U: amount of heap currently used
F: amount of heap currently free
α: scan time / mark time (<< 1)

Basic Algorithm: Mark and scan

- Allocate memory as fixed size cells
- When all memory has been allocated:
  1. Pause execution
  2. set alloc array to all false
  3. Traverse all live data (in active function stack frames) setting alloc[i] to true for each cell i traversed.
     - Terminate traverse on cells with alloc already true
  4. Scan through memory gathering all cells i with alloc[i] false in a single free list of memory
  5. Restart execution

garbage collection

Basic Algorithm: Copying

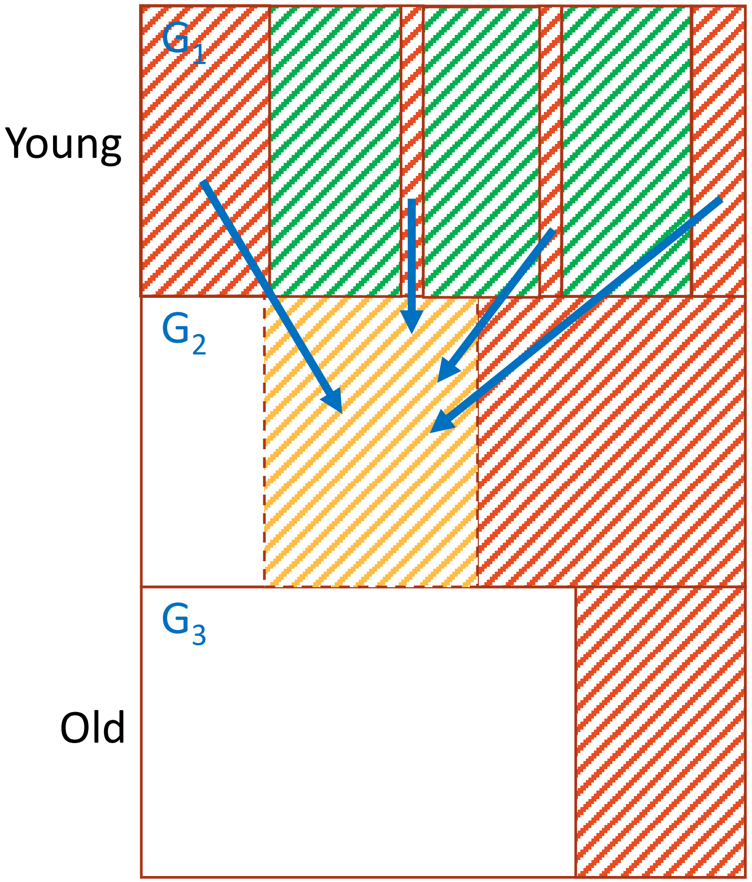
- Divide memory into two equal sized areas: A and B
- Allocate memory as variable size blocks from one area only (say A)
- When all memory in A has been allocated:
  1. Pause execution
  2. Traverse all live data (in active function stack frames) copying cells into B memory. Overwrite A cells with indirection to B after copying. Terminate traverse on reaching indirection (since after that all will have been copied)
  3. All data is now in B, compacted, and A is empty.
  4. Restart execution - next time round do copy from B back to A.

garbage collection

Generational Copying

Further information

- Copying becomes more efficient as F increases relative to U.
- Clever idea to decrease U.
  - Divide all heap data into **generations**
  - G_1 contains all new allocated data
  - If data stays resident for more than one GC cycle: copy it from G_n to G_{n+1}
  - Typically use 2 or 3 generations but number can increase as needed
  - Garbage collect each generation separately only when it runs out of free space
    - Older generations have data that changes less and are garbage collected less often



Snapshot of memory with generational copying gc when G1 is full