COP 4555 Final Exam Notes

Wednesday, June 18, 2014 9:30 PM

OO Programming

The 3 key factors in OO programming:

1. Encapsulation (data hiding)
2. Inheritance
3. Dynamic method binding
   a. You don't know at compile time what type the object referred to by a variable will be at run time.
4. Data abstraction

SMALLTALK is canonical OO language
Generics are for abstracting over types.
Non-virtual functions require no space at run-time.

F#

No automatic type coercion. Must use explicit type casting.
Functions are **first class values**
-like high-order functions

Curried Functions:	t1 -> (t2 -> t3)

• For example f(x,y) can be computed as f1(x) which returns a function f2, and applying f2(y) returns f(x,y). Thus f(x,y)

*= f1(x)(y). f has a type: X * Y -> Z*
*> let time (x, y) = x * y;;*
*val time : x:int * y:int -> int*

An equivalent curried version:
*> let ctime x y = x * y;;*
val ctime : x:int -> y:int -> int
> ctime 4;;
val it : (int -> int) = <fun:it@6-1>
> it 5;;
val it : int = 20

Additional ways of defining curried functions:
*> let ctime1 x = fun y -> x * y;;*
val ctime1 : x:int -> y:int -> int
*> let ctime2 = fun x -> (fun y -> x * y);;*
val ctime2 : x:int -> y:int -> int

Type associates to the right. Ex: t1 -> (t2 -> (t3 -> t4))
Function applications associate to the left

Using function **compositions**
let fact = downFrom >> prod;;
Using **forward pipeline operator**
let fact n = n |> downFrom |> prod;;

*List.map (fun n -> n*n) [1;2;3]@[4;5;6] is parsed as*
*((List.map (fun n -> n*n)) [1;2;3])@[4;5;6]*
val it : int list = [1; 4; 9; 4; 5; 6]

```
let rec map f = function
    | [] -> []
    | x::xs -> f x :: map f xs;;
val map : f:('a -> 'b) -> _arg1:'a list -> 'b list
```

Checklist for Recursive Programming
1) Make sure that each base case returns the correct answer,
2) Make sure that each non-base case returns the correct answer, assuming that each of its recursive calls return the correct answer,
3) Make sure that each recursive call is on a smaller input. (A partial order ≤ is a binary relation on a set S: reflexive, transitive, anti symmetric. ≤ is well-founded if every non-empty subset of S has a minimal element). Correctness of the checklist: it is a concrete way of realizing mathematical induction

Analyzing Code with the Checklist
1) Is there any situation in which a base case fails to return the correct result for the input?
2) Is there any situation in which the code for a non-base case fails to transform correct results returned by the recursive calls into correct result for the input?
3) Is there any situation in which the definition can make a recursive call on an input that is not smaller than the original input?

Functional Languages

Alonzo Church
F# inspired by lambda calculus
-No mutable state-No side effects
-1st class and **high-order functions**
-take a function as an argument, or return function as a result - great for building things
Pure Lisp is purely functional
Strict language - requires all arguments to be well-defined, so applicative order can be used.
Non-strict language - does not require all arguments to be well-defined; it requires normal-order evaluation
Advantages of functional languages
-lack of side-affects makes programs easier to understand
-lack of explicit evaluation order offers possibility of parallel evaluation
-lack of side effects and explicit evaluation order simplifies some things for a compiler
-programs are often surprisingly short
-language can be extremely small and yet powerful
Disadvantages of functional languages
-difficult to implement efficiently on von Neumann machines
-lots of copying data through parameters
-heavy space use for recursion
-requires garbage collection

More F#

Defining Records:
type RecordName = {field1: type1; field2: type2; …}
let arecord = {field1 = value1; …; fieldn = valuen}
type Person = {Name:string; Age:int}
type Person = {Name: string; Age: int;}
let p = {Age = 15; Name = "David"}
val p : Person = {Name = "David"; Age = 15;}

p.Name;;
val it : string = "David"

Discriminated Union Types
type name = id1 | … | idn
type color = Red | Green | Blue;;

```
type color =
    | Red
    | Green
    | Blue

let opinion = function
    | Blue -> "A"
    | Green -> "B"
    | Red -> "C"
val opinion : _arg1:color -> string
> opinion Red;;
val it : string = "C"
```

Binary Search Tree
type 'a tree = Lf | Br of 'a * 'a tree * 'a tree;;

```
type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree
> Lf;;
val it : 'a tree = Lf

> Br (10, Lf, Lf);;
val it : int tree = Br (10,Lf,Lf)

let rec element n = function
    | Lf -> false
    | Br(m, t1, t2) -> if n = m then true
                       elif n < m then element n t1
                       else element n t2
val element : n:'a -> _arg1:'a tree -> bool when 'a : comparison
```

```
let rec insert n = function
    | Lf -> Br(n, Lf, Lf)
    | Br(m, t1, t2) -> if n < m then Br(m, insert n t1, t2)
                       else Br(m, t1, insert n t2)
val insert : n:'a -> _arg1:'a tree -> 'a tree when 'a : comparison

let rec buildtree = function
    | [] -> Lf
    | x::xs -> insert x (buildtree xs);;
val buildtree : _arg1:'a list -> 'a tree when 'a : comparison

let rec sum = function
    | Lf -> 0
    | Br(m, t1, t2) -> m + sum t1 + sum t2
val sum : _arg1:int tree -> int
```

Notice that to process values of a discriminated union, we must use pattern matching

• Notice that to process values of a discriminated union, we must use pattern matching
• Also notice that, as was the case with list operations, these operations on trees are all non-destructive.
• For instance, insert n t returns the result of inserting n into t, without destroying t. But, because of sharing, this can be done efficiently.

type mix = Int of int | Str of string | Boo of bool;;

```
type mix =
    | Int of int
    | Str of string
    | Boo of bool
> [Int 5; Boo true; Str "abc"; Int 10];;
val it : mix list = [Int 5; Boo true; Str "abc"; Int 10]
```

type 'a stream = Cons of 'a * (unit -> 'a stream);;
• An 'a stream thus has the form Cons(x, xsf), where x is a the head of the stream, and xsf is a function that can be called to give the tail of the stream

let rec upfrom n = Cons(n, fun () -> upfrom(n+1));;

F# type inference does not support subtyping of numeric types. For example, F# does not regard int as a subtype of float.
• Overloaded numeric operators are another source of trouble for F# type inference. F# demands that each occurrence of such an operator be given a unique type or defaults to int type:

Chapter 12: Concurrency

A **process** or **thread** is a potentially-active execution context.
A process can be thought of as an abstraction of a physical PROCESSOR
Processes/Threads can come from :
– multiple CPUs – kernel-level multiplexing of single physical machine – language or library level multiplexing of kernel-level abstraction
• They can run :
– in true parallel – unpredictably interleaved – run-until-block
Two main classes of programming notation:
– synchronized access to shared memory
– message passing between processes that don't share memory
Race conditions
– A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen
– Race conditions are not all bad; sometimes any of the possible program outcomes are ok (e.g. workers taking things off a task queue)
– **SYNCHRONIZATION** is the act of ensuring that events in different processes happen in a desired order
– Synchronization can be used to eliminate race conditions
-Most synchronization can be regarded as either
• **Mutual exclusion** (making sure that only one process is executing a CRITICAL SECTION [touching a variable, for example] at a time), or as
• **CONDITION SYNCHRONIZATION**, which means making sure that a given process does not proceed until some condition holds (e.g. that a variable contains a given value)
Mutual exclusion is not a form of condition synchronization
– The distinction is basically existential v. universal quantification
    • Mutual exclusion requires multi-process consensus
• We do NOT in general want to over-synchronize
    – That eliminates parallelism, which we generally want to encourage for performance
• Basically, we want to eliminate "bad" race conditions, i.e., the ones that cause the program to give incorrect results

To implement synchronization you have to have something that is **ATOMIC**
• that means it happens all at once, as an indivisible action
• In most machines, reads and writes of individual memory locations are atomic
SCHEDULERS give us the ability to "put a thread/process to sleep" and run something else on its process/processor
– start with coroutines
– make uniprocessor run-until-block threads
– add preemption
– add multiple processors
Coroutines
– Multiple execution contexts, only one of which is active
Run-until block threads on a single process
– Need to get rid of explicit argument to transfer
– Ready list data structure: threads that are runnable but not running procedure reschedule:
    t : thread := dequeue(ready_list)
    transfer(t)
Preemption on a Uni-Processor
– Use timer interrupts (in OS) or signals (in library package) to trigger involuntary yields
Condition synchronization with atomic reads and writes is easy • Mutual exclution is harder

Repeatedly reading a shared location until it reaches a certain value is known as **SPINNING or BUSY-WAITING**
• A busy-wait mutual exclusion mechanism is known as a **SPIN LOCK**
    – The problem with spin locks is that they waste processor cycles
    – Sychronization mechanisms are needed that interact with a thread/process scheduler to put a process to sleep and run something else instead of spinning
    – Note, however, that spin locks are still valuable for certain things, and are widely used
        • In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead
SEMAPHORES were the first proposed SCHEDULER-BASED synchronization mechanism, and remain widely used. A semaphore is a special counter. Keeps track of the difference between the number of P and V operations that have occurred.
• **CONDITIONAL CRITICAL REGIONS** and **MONITORS** came later
Problems with semaphores
– They're pretty low-level.– Their use is scattered all over the place
Monitors were an attempt to address the two weaknesses of semaphores listed above
A monitor is a shared object with operations, internal state, and a number of condition queues.
Only one operation of a given monitor may be active at a given point in time
• A process that calls a busy monitor is delayed until the monitor is free
JAVA
In Java, every object accessible to more than one thread has an implicit mutual exclusion lock
Within a synchronized statement or method, a thread can suspend itself by calling the predefined method **wait** with no argument, which needs to be within a condition testing loop.
To wake up all threads waiting on a given object, Java provides **notifyAll** method.
Lock Variables, provided in java.util.concurrent package provides a more general solution to synchronization.

Java Memory Model:
1) specifies exactly which operations are guaranteed to be ordered across threads;
2) specifies for every pair of reads and writes in a program execution, whether the read is permitted to return the value written by the write. A read is allowed to return values only from unordered writes or from immediately preceding ordered writes
2) specifies for every pair of reads and writes in a program execution, whether the read is permitted to return the value written by the write. A read is allowed to return values only from unordered writes or from immediately preceding ordered writes

More F#

The F# **value restriction** says that when we have a declaration
let x = e;;
we can give x a polymorphic type only if e is something called a **syntactic value**.
A syntactic value is an expression that can be evaluated without doing any computation.
• **Syntactic values** include:
1. literals and identifiers (e.g. 3, n)
2. function expressions (e.g. (fun n -> n))
3. constructors applied to syntactic values (e.g. (12, x::[true]))
Syntactic values do not include function calls:
let x = List.rev [];;

Following gives error:
let revlists = List.map List.rev;;
Fix with:
let revlists = (fun xs -> List.map List.rev xs);;
val revlists : xs:'a list list -> 'a list list
Trick is known as **eta expansion**.

Mutable reference cells (using **ref** command)
let r = ref 20;;
val r : int ref = {contents = 20;}
Retrieve with
> !r;;
val it : int = 20
Update with
> r := !r + 2;;
val it : unit = ()
> !r;;
val it : int = 22

> let s = r;;
val s : int ref = {contents = 22;}
> let t = ref (!r);;
val t : int ref = {contents = 22;}
r and s are aliases, and t is a different cell

More Chapter 12

A Java thread is allowed to buffer or reorder its writes until the point at which it writes a **volatile** variable (its value can be changed by other threads) or leaves a monitor (releases a lock, leaves a synchronized block, or wait).

The compiler is free to:
1) reorder ordinary reads and writes in the absence of intrathread data dependences;
2) move ordinary reads and writes down past a subsequentvolatile read;
3) move ordinary reads and writes up past a previous volatile write;
4) move ordinary reads and writes into a synchronized block from above or down;
5) but cannot reorder volatile accesses, monitor entry or monitor exit with respect to one another.

A Java thread can be in one of the following states:
• New – A thread is just instantiated.
• Runnable – a thread is executing in the JVM.
• Waiting – a thread is waiting indefinitely for other threads to perform actions.
• Timed_Waiting (sleeping) – a thread is waiting for other threads to perform actions for up to a specified waiting time.
• TERMINATED (dead) – a thread has exited.

Synchronization is a mechanism used to prevent the above errors, but may introduce thread contention errors – deadlock (multiple threads are waiting circularly, thus everyone stuck) and livelock (some thread cannot make any progress) if not used carefully;

Lock objects support locking idioms that simplify many concurrent applications.
Executors define a high-level API for launching and managing threads.
Concurrent collections make it easier to manage large collections of data
Atomic variables have features that minimize synchronization and help avoid memory consistency errors.