CSE 505, Fall 2006, Assignment 5
Due: Friday 8 December 2006, 4:30PM

Last updated: November 28

You also need `oo.ml` and `cml.mli`, available on the course website.

1. (Concurrent ML) You will implement the two interfaces in `cml.mli` using Concurrent ML. Do *not* use mutable references, mutexes, condition variables, fork-join, etc. All you need is thread-creation and the Concurrent ML primitives provided in the `Event` module. Remember to compile `cml.ml` via `ocamlc -vmthread -o hw5cml threads.cma cml.ml` or similar. Also note Caml terminates when the main thread terminates, so it is sometimes convenient when testing to put in an infinite loop (preferably with a call to `Thread.yield`) in the main thread.

   (a) Define the type `barrier` and the functions `new_barrier` and `wait`. If a barrier is created by `new_barrier` i and a thread makes one of the first i calls to `wait`, then the thread should block until the i^{th} call, at which point all i blocked threads should proceed. If a thread calls `wait` after there have been i calls, it should block forever.

   How to do it: `new_barrier` should return a channel that a newly created thread receives on. `wait` should send on this channel another channel and then receive on the channel it sends. (It does not matter what is sent on this channel; `()` is a fine choice.) The newly created thread should use the arguments of a tail-recursive function to remember how many waiters there are and what channels to send on after the last arrives. The thread can then terminate. Note you do not need `choose` or `wrap`.

   (b) Define the type `funny_counter`, which is like a type we considered in lecture 13 except it is imperative instead of functional. The function `new_counter` creates a new counter whose "sum" and "length" are both 0. `add` increases the sum of its second argument by its first argument and the length of its second argument by 1. `sum` returns its arguments current sum. `sum_of_longer` returns the sum of whichever argument has a greater length (either is fine if the lengths are the same). Note clients of this interface *cannot* directly retrieve a counter's length.[1]

   How to do it: For each counter, create a thread with an infinite loop that uses two channels, one for receiving a value to add and one for sending the current sum *and* length. "Store" the sum and length via arguments to a tail-recursive function. Have `add`, `sum`, and `sum_of_longer` communicate with the thread via the channels. These functions can get a counter's current length; it is clients of the interface in `cml.ml` that cannot. Note you do need `choose`, and using `wrap` avoids clumsy workarounds so use it.

   (c) In a short English paragraph, explain how using separate channels for the sum and length in part (b) could lead to a race condition.

2. (Implementing Class-Based OOP) You will complete the code in `oo.ml`, which demonstrates one way to implement simple object-oriented programming. There are *two* similar languages involved.

   Our "source" language is an ML expression of type `((classname option) classdef list) * exp` i.e., a pair of (1) a list of class definitions where the `super` field contains a `classname option` and (2) a "main" expression. The `super` field is the immediate superclass if there is one. You may *assume* all the following: class names are distinct, the superclass relation has no cycles, the methods in one class definition have distinct names, the fields in one class definition have names distinct from each other and from all fields in all superclasses. A subclass extends its superclass (if any) by adding all its fields and any methods with names different from those in any superclass. To override a method, a class definition just includes a method of the same name (see the extra credit). The semantics of methods and expressions should be straightforward; ask if something is unclear.

   The language for our interpreter is an ML expression of type `((classname list) classdef list) * exp`. Unlike the source language:

[1] Unfortunately, your instructor did come up with a way to figure out the length with this interface, but it is slow, clumsy, and not the point of the problem.

- The `super` field holds a list of all superclasses (transitively). There may be none.
- The `methods` field holds all the methods for a class, including those defined in superclasses.
- The `fields` field holds all the fields for a class, including those defined in superclasses.

(a) The code given to you defines a `boolclass` (in the source language). Define two subclasses `trueclass` and `falseclass`. Each should override the `ifThenElse` method, using the encoding of true returning the first of two objects and false the second.

(b) Implement `translate_classdefs` of type `(classname option) classdef list -> (classname list) classdef list` for translating a list of class definitions from the source language to the interpreter language. The sample solution uses several helper functions and several functions in Caml's list library, and is 30–35 lines.

(c) Complete the interpreter `interp` of type `classname list classdef list -> (varname * valu) list -> valu -> exp -> valu`. Three cases are done for you. The other cases are described informally below; there should be no surprises. In general, you may throw whatever sort of exception you like for bad programs, even `List.Not_found`. The arguments to `interp` are the (translated) class definitions (needed for method calls), an environment (needed for local variables), the value currently bound to self, and the expression being evaluated. Note `New` creates a `valu` where the fields start uninitialized.

- Getting a field fails if the result of evaluating the subexpression does not have that field or that field is still uninitialized. Otherwise, it returns the `valu` held in the field.
- Setting a field fails if the result of evaluating the first subexpression does not have that field. Otherwise, it does the appropriate mutation.
- A let-expression is interpreted very much like in ML.
- A cast fails if the class of the result of evaluating the subexpression is neither the class specified nor a (transitive) superclass of it. Otherwise, the result is the result of the subexpression.
- A local variable is looked up in the environment.
- A method call evaluates all the subexpressions and then looks up the method in the class of the receiver. The lookup just uses the name of the method, returning the first method in the list with that name. The result is the evaluation of the method body using the receiver for self and an environment containing bindings just for the method arguments. It is an error to have the wrong number of arguments.

(d) In a short English paragraph, explain why having casts in our language is of little use. In particular, give *two* standard uses of casts in languages like Java and C++, one of which affects type-checking and one of which affects method-call semantics. Explain how neither is relevant with our implementation.

(e) **(Extra Credit)**

- (Fairly easy) Change the interpreter to support "arity overloading" — a method call should not fail due to having the wrong number of arguments if a superclass of the receiver defined a method with the same name and the correct number of arguments.
- (Fairly easy) Change your interpreter to detect type mismatches on field-assignment or method-call and fail immediately.
- (Much more challenging) Changing your translation but not your interpreter, implement static overloading. Document ambiguities and how you deal with them. (Small hints: `Fail` causes ambiguities. You will have to implement a type-checker even though the language will not be type-safe.)

What to turn in: Hard-copy (written or typed) answers to problems 1c and 2d, and Caml source code in files `oo.ml` and `vote.ml`.

Email your source code to Anna as `firstname-lastname-hw5.tgz` or `firstname-lastname-hw5.zip`. The code should untar/unzip into a directory called `firstname-lastname-hw5`. Hard copy solutions should be put in Anna's grad student mailbox or given to her directly.