UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

**CS 263** **Spring 1996**
**K. A. Yelick**

# CS 263: Design of Programming Languages
# Homework 6
# Due May 16, 1996

## 1. Semantics of Concurrency

Kay Continuation of Complex Denotation Corporation is preparing an article on FL! coroutines for Transactions on Crufty Stuff. Kay has extended FL! with two coroutine constructs.

```
E ::= ...                    Usual FLK! expressions
      | (coroutine E1 E2)
      | (wait)
```

Informally, `(coroutine E1 E2)` creates two control threads: one for evaluating `E1`, the other for evaluating `E2`. Only one thread actively executes at a time. Initially, the `E1` thread is active, and the other thread is suspended. If execution of the active thread results in a value or error, `coroutine` returns the same value or error. If a `(wait)` is executed, the active thread is suspended, and the suspended thread starts executing. To suggest the ping-pong control flow, Kay has christened her new dialect "FLing-FLong."

Figure 1 shows the first draft of FLing-FLong semantics. Kay has introduced the new semantic domain *Thread* for modelling threads of control and has modified the definition of *Cmdcont*. A thread is a function that takes a store and another thread and returns an answer. Calling thread $t_{act}$ on a store and suspended thread $t_{susp}$ effectively executes $t_{act}$ in a context where `(wait)` will activate $t_{susp}$ after suspending $t_{act}$. Kay assumes that all the semantic domains in FLK! refer to the new definition for *Cmdcont*. The extra *Thread* argument in $\mathcal{E}$, *Expcont*, *Cmdcont* and *Procedure* models the suspended control thread within a `coroutine`.

$$
\begin{array}{lcccl}
t & \in & \mathit{Thread} & = & \mathit{Store} \to \mathit{Thread} \to \mathit{Answer} \\
c & \in & \mathit{Cmdcont} & = & \mathit{Store} \to \mathit{Thread} \to \mathit{Answer}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}[\![\texttt{(coroutine}\ E_1\ \ E_2\texttt{)}]\!] \\
\ = \lambda u k s_0 t_{outer}\ .\ \textbf{let}\ \mathit{restore\&return}\ \textbf{be}\ (\lambda v s_1 t_{inner}\ .\ \ (k\ v\ s_1\ t_{outer}))\ \ \textbf{in} \\
\qquad\qquad\qquad (\mathcal{E}[\![E_1]\!]\ u \\
\qquad\qquad\qquad\quad \mathit{restore\&return} \\
\qquad\qquad\qquad\quad s_0 \\
\qquad\qquad\qquad\quad (\mathcal{E}[\![E_2]\!]\ u\ \mathit{restore\&return}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{top\text{-}level\text{-}cont} : \mathit{Expcont} = \lambda v s t\ .\ (\mathit{Value}\mapsto \mathit{Answer}\ v) \\
\mathit{error\text{-}thread} : \mathit{Thread} = \lambda s t\ .\ (\mathit{Error}\mapsto \mathit{Answer}\ \mathit{unit}) \\
\mathit{error\text{-}cont} : \mathit{Cmdcont} = \lambda s t\ .\ (\mathit{Error}\mapsto \mathit{Answer}\ \mathit{unit})
\end{array}
$$

Figure 1: FLing-FLong semantics

Kay has been called to Washington to study speed limits on the Information Autobahn and left you to work on the paper.

a. Fill in the missing definitions for $\mathcal{E}[\![\texttt{(wait)}]\!]$ and $\mathcal{T}[\![E]\!]$.

b. What are the values of the following FLing-FLong expressions?

i.
```
(let ((a (ref 1)))
  (let ((f (lambda (x)
             (begin (:= a (* x (^ a)))
                    (wait)
                    (:= a (+ x (^ a)))
                    (^ a)))))
    (coroutine (f 2) (f 3))))
```

ii. `(begin 1 (wait) 2)`

iii. `(coroutine (wait) (wait))`

iv.
```
(coroutine (begin (wait) 2)
           (coroutine (begin (wait) 3)
                      (begin (wait) 4)))
```

c. Kay wants to add the construct `(disable-wait` $E$`)` which evaluates $E$ in such a way that all `(wait)` expressions within $E$ return *unit* immediately without activating the suspended thread. The value of the `disable-wait` expressions is the value of $E$. Give the valuation clause $\mathcal{E}[\![\texttt{(disable-wait}\ E\texttt{)}]\!]$.

d. Kay decides to make the communication between coroutines more explicit by replacing `(wait)` with `(send` $E$`)` and `(receive` $E$`)`. `(send` $E$`)` evaluates $E$ and sends the value to the suspended coroutine. The value of the `send` expression is *unit*. In `(receive` $E$`)`, $E$ must evaluate to a *Location* which is assigned the value sent by the sibling coroutine; the value of `receive` is also *unit*. Write the valuation clauses for `send` and `receive`. Does your solution handle mismatched communication points?

**2. Types**

Suppose the following program were translated into Scheme/XSPA. What is the most general explicit type of the procedure `filter`?

```
(define convert-filter
      (lambda (pred cvt lst)
           (if (null? lst)
                lst
                (if (pred (car lst))
                    (cons (cvt (car lst)) (filter pred (cdr lst)))
                    (filter pred (cdr lst))
                 ))))
```

**3. Data Abstraction and Inheritance**

This question asks you about subverting the intended behavior of an object, or class of objects, through subtyping. We will discuss this using an example class mechanism that could be added to ML.

Our simple class mechanism will have class declarations and object creation operations of the form

```
class t(<args>)=
private
  <decls>
public
  <decls>
end;

val x = new t(<args>);
```

where it is assumed that any initialization can be carried out within the "private" part of the class body. Essentially, these "Object-ML" declaration and create operations will have the same semantics as the following function declaration and call:

```
fun new_t(<args>)=
let
  <decls>
in
  {
   <decls>
  }
end;

val x = new_t(<args>);
```

where the "private" declarations go before the `in` and "public" ones after. In addition, we will use "dot" notation for sending a message. For example, a priority queue closure could be defined and manipulated as follows.

```
exception Empty;
class pq() =
```

```
    private
        val contents = (ref nil)  : int list ref
        fun insert(x,nil)   = [x]
        |   insert(x, y::l) = if x<y then x::(y::l)
                                     else y::insert(x,l)
    public
         empty()   = !contents = nil,
         insert(x) = (contents := insert(x,!contents)),
         delete()  = case !contents of
                              nil => raise Empty
                            | y::l => (contents := l; y)
    end;

val h = new_pq();
h.insert(2);
h.insert(0);
h.insert(1)
h.delete();
```

The value of the final expression, `h.delete`(), would be 0, with 1 and 2 remaining in the priority queue.

Aside from syntactic differences between this "class" construct and ML functions returning closures, the reasons for adding this construct to ML would be to permit subtyping and inheritance. One possibility would be to allow a subclass declaration of the form illustrated by the following example subclass of `pq`.

```
class balanced_pq() = extend pq() with
    private
        fun median(nil) = raise Empty
        |   median([x]) = x
        |   median(x::l) =  (* code omitted *)
    public
         middle()  = median(!contents)
    end;
```

A `balanced_pq` object will be like a `pq` object, but with one additional method (`middle`) returning the median element of the priority queue. Note that unlike C++, the public function of a subclass is allowed to access the private data of the superclass. In this example, this shows up in the implementation of `middle`, which accesses the private variable `contents`. (Those of you familiar with C++ might prefer to think of the `private` part of a class as `protected` rather than `private`.)

**Question A:** The visibility rules are that any private data or function declared in a superclass may be accessed in any subclass. Explain how a "devious" subclass could add a method that causes priority queues to behave incorrectly.

**Question B:** The rules in the Liskov and Wing paper and intended to prevent this kind of behavior. Which of their rules is being violated in your example?