Administrative Notes

Class on Thursday will be in 373 Soda at the usual time – the room is being used for a faculty candidate seminar.

Midterms: Grading is underway.

Control Example

Unusual cases or errors arise in code. They can be treated using:

- Wrapping code in a conditional
- Using a goto
- use a (local) exit

If effects of these errors propagate beyond a procedure call:

- return error flag that is checked by each statement
- use an exception handling mechanism
- pass "rest of program as a function" and call it at the error point

Control Examples

If functions have multiple return values:

- use two different procedures
- package the results into a list or other structure
- modify the input parameters (for stateful languages)
- pass a function to handle the multiple return values

Multiple processes used to organize programs in OS, etc., e.g. produce/consumer processes

- use coroutines
- user higher order functions and recursion

Coroutines

```
program P;        coroutine A;        coroutine B;
...               ...                 ...
call A;           resume B;           ...
...               ...                 resume A;
end               resume B;           ...
                                      return
```

- `resume` transfers control from one coroutine to another
- control within a coroutine always start from the place it last left off (the beginning if it has not yet run)
- semantics are not true concurrency, but interleaving between routines
- coroutines were first described by Conway in 1963 CACM paper. The exist in Simula, Modula-2, and Ada.

Coroutines with Higher Order Functions

The effect or coroutining can be simulated using higher order functions. The following code adds 5 numbers starting with 3 (3+4+5+6+7 = 25).

```
(define count-from
 (lambda (num)
  (letrec ((new-prod
     (lambda (n)
      (lambda (consume)
       (consume n (new-prod
          (+ n 1)))))))
  (new-producer num))))
```

```
(define add-first
 (lambda (count)
  (letrec ((new-consumer
     (lambda (c)
      (lambda (value next-prod)
       (if (= c 0)
           0
           (+ value
              (next-prod
               (new-consumer
                (- c 1))))))))))
    (new-consumer count)))
```

```
((count-from 3) (add-first 5))
```


Semantics of Control

Consider slightly simpler case: a `break` statement.

```
(sequence E1 break E2)

(sequence (if B break #u) (:= x 2) E3 ... En)
```

How might you give an operational semantics for such a construct?

Why is this harder than giving the semantics of conditionals?

What about a denotational semantics in the (direct) style we have been using?

Another style of denotational semantics is called "standard" semantics. It involves an explicit notion of control, called continuations.

Continuations

A *continuation* is a function that represents "the rest of the program."

- In a statement based language, this is the abstract program counter (plus 1).
- In an expression-based language, or in the middle of an expression in general, one also needs the value of the expression so far.

```
x := x * x;
if (b < 0)
  then
    y := x;
  else
    y := 0;
```

```
(let ((square (lambda (x)
               (* x x))))
  (+ (square 5)
     (* (+ 1 2)
        (square 5))))
```

One continuation in the 2nd example is:

```
(lambda (v1) (+ v1 (* (+ 1 2) (square 5))))
```

In some languages, e.g., Scheme, there is a linguistic construct to create continuations – `cwcc`, which stands for "call with current continuation."

Standard Semantics

Direct semantics: use only environment and store (for stateful languages) domains as inputs to the valuation function.
Standard semantics: use an additional continuation parameters to the valuation function.
There are two continuation domains:

- *command continuation*

$$\gamma \in \mathit{Cmdcont} = \mathit{Store} \rightarrow \mathit{Answer}$$

- *expression continuation*

$$\begin{aligned} k \in \mathit{Expcont} &= \mathit{Expressible} \rightarrow \mathit{Cmdcont} \\ &= \mathit{Expressible} \rightarrow \mathit{Store} \rightarrow \mathit{Answer} \end{aligned}$$

The *Answer* domain represents the final value of the program, which is typically an expressible value.

Changes for Standard Semantics

The valuation function for expression has changed (as promised):

$\mathcal{E} : \text{Exp} \rightarrow \mathit{Environment} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}$
or, equivalently:
$\mathcal{E} : \text{Exp} \rightarrow \mathit{Environment} \rightarrow \mathit{Expcont} \rightarrow \mathit{Store} \rightarrow \mathit{Answer}$

For example, an expression of the form:

$$(\mathcal{E}[\![E]\!]e(\lambda v \; . \; \square))$$

means: find the value for E in the environment e and name the result v in $\square$. To change a store s_0 into another s_1, use:

$$(\mathcal{E}[\![E]\!]e(\lambda v \; s_1 \; . \; \square) \; s_0)$$

And *test-xxx* functions convert continues expecting type *xxx* into an expression continuation, but unlike the *with-xxx*, these do not propagate errors.

Top-level Valuation Function for FLK!

The top-level valuation function for the whole program.

$$\mathcal{TL} : \text{Exp} \rightarrow \mathit{Answer}$$

$$\mathcal{TL}[\![E]\!] = (\mathcal{E}[\![E]\!]\mathit{empty\text{-}envtop\text{-}level\text{-}contempty\text{-}store})$$

$$\begin{aligned}&\mathit{top\text{-}level\text{-}cont} : \mathit{Expcont}\\ &\quad = \lambda v \; . \; \lambda s \; . \; (\mathit{Value} \mapsto \mathit{Expressible} \; v)\end{aligned}$$

$$\mathcal{E}[\![(\texttt{begin} \;\; E_1 \; E_2) = \lambda ek \; . \; (\mathcal{E}[\![E_1]\!]e \; (\lambda v_{ignore} \; . \; (\mathcal{E}[\![E_2]\!]e \; k)))$$

For example:

`(begin E1 E2)`

Take top-level (id) continuation, apply meaning of E2 and then apply meaning of E1.

Valuation Functions for Simple Expressions

Here is the valuation function for identifiers. Symbols and built-in function are similar.

$$\mathcal{E}[\![I]\!] = \lambda ek \; . \; (\mathit{ensure\text{-}bound}(\mathit{lookup} \; e \; I) \; k)$$

where *ensure-bound* checks the binding:

$$\begin{aligned}&\mathit{ensure\text{-}bound} : \mathit{Binding} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}\\ &= \lambda \beta k \; . \; (\textbf{matching}\beta\\ &\qquad \triangleright (\mathit{Denotable} \mapsto \mathit{Binding} \; v) \; \| \; (k \; v)\\ &\qquad \triangleright (\mathit{Unbound} \mapsto \mathit{Binding} \; \mathit{unbound}) \; \|\\ &\qquad\qquad (\mathit{error\text{-}cont} \; \texttt{unbound-variable})\\ &\qquad \textbf{endmatching}\end{aligned}$$

Functions in Standard Semantics

The *Procedure* domain is also changed from the direct semnatics:

$$p \in \mathit{Procedure} = \mathit{Denotable} \rightarrow \mathit{Expcont} \rightarrow \mathit{Cmdcont}$$

$$\begin{aligned}\mathcal{E}[\![(\texttt{call} \;\; E_1 \; E_1)]\!] = \lambda ek \; . \; (\mathcal{E}[\![E_1]\!] \; e \; (\mathit{test\text{-}procedure}(\lambda p \; . \; (\mathcal{E}[\![E_2]\!] \; e\\ (\lambda v \; . \; (p \; v \; k))))))\end{aligned}$$

$$\begin{aligned}\mathcal{E}[\![(\texttt{lambda} \;\; I \; E)]\!] = \lambda e \; k_1 \; . \; (k_1(\mathit{Procedure} \mapsto \mathit{Value}(\lambda \delta k_2 .\\ (\mathcal{E}[\![E]\!][I : \delta]e \; k_2))))\end{aligned}$$

Non-Local Exits

Add to FLK!

`(label I E)` establish a named label that can be a jump point

`(jump E1 E2)` jump to E1 (which must be a jump point) and pass value of E2

```
(+ 1 (label lab1 (+ 2 (* 3 (/ 4 (if b (jump lab1 5) 1))))))

                  b = #t, evaluates to:
                  b = #f, eavluates to:
```


Exit Semantics

Control-point = Expcont

$v \in$ *Value* = ... + *Control-point*

$\mathcal{E}$[[`(label I E)`]] = $\lambda\, u\, k\, .\, (\mathcal{E}$[[E]] (extend-env u I (Control-point->Value k)) k)

$\mathcal{E}$[[`(jump E1 E2)`]] = $\lambda\, u\, k0\, .\, (\mathcal{E}$[[E1]] u ($\lambda\, v1$. case $v1$

> (Control-point->Value k1) ||

($\mathcal{E}$ [[E2]] u k1)

> else error-cont

endcase))

Exceptions in Other Languages

In CLU there are two mechanism for "abnormal" control flow. One is the exit statement:

except when name $_1$: val $_{1,1}$,.., val $_{1,i}$

. . .

when name $_n$: val $_{n,1}$,..., val $_{n,j}$

exit: drop out of the current execution to the nearest enclosing handler

```
begin
   while i <= high do
      x := hairy(a[i])
      if special(x) then exit found (x) end
      i := i+1
      end
   x := make_new_one()
   end except when found (j): do_something(j) end
```


Exceptions in Other Languages

Common Lisp has a similar mechanism;

(block name E1...En) - like begin (or sequence), but marks a jump point

(return-from name E) - jumps to the "name" jump point, passing E as the block result

```
(block one                                  => 3
  (block two
    (return-from one 3) 2) 1)
```

Both languages: exits are statically scoped.

In CLU, they do not propagate outside a procedure call, but in Common Lisp they may:

```
(block one
  ((lambda ()
      (return-from one 3) 2))         => 3
  1)
```

Exceptions Questions

Scoping of exception names:

1) static
2) dynamic
3) 1 level dynamic and then static

Control:

1) termination semantics
2) resumption semantics
3) dismissal (referral)

Exception values:
1) none 2) one 3) arbitrary

Which languages use these? Why?