Lecture 17: Control Flow

Steven Holtzen
s.holtzen@northeastern.edu

CS4400/5400 Fall 2024

1 What is control flow?

- **Control flow** determines which part of the program executes next.
- So far, all of the programming languages we have seen have had simple control-flow structure. We have seen two syntactic constructs that give the programmer control over the control flow of a program:
  - Function calls.
  - If expressions.
- Practical programming languages often have more interesting ways of modifying control flow, for example exceptions. Consider the following Java code:

```
public class Main {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

- This code does not simply execute one expression after the other in sequence like all of our interpreters so far; it relies on the ability to jump from one place of execution to another
- This module is about implementing languages with richer control flow constructs than the ones we have seen so far. To do this, we will rely heavily on an idea called a continuation, which we will see today.

2 The Control Context

- Thus far, we have seen how we can write programs that manipulate names and mutable state by enriching our interpreter with heaps and environments.
- Now, we will enrich our interpreters with a notion of control context in order to implement interesting control-flow structure
- A **control context** tracks which operations remain to be performed in order to finish a computation. If you're familiar with the notion, a control context is analogous to a call stack.
- As an example, consider the following definition of a factorial function (written in OCaml syntax):

```
let rec fact n =
  if n = 0 then 1 else n * (fact (n - 1))
```

- Let's step through evaluating the recursive call `(fact 4)`:

```
fact 4
--> 4 * (fact 3)
    ^^^ control context
--> 4 * 3 * (fact 2)
    ^^^^^^^ control context
--> 4 * 3 * 2 * (fact 1)
    ^^^^^^^^^^^ control context
--> 4 * 3 * 2 * 1 * (fact 0)
    ^^^^^^^^^^^^^^^ control context
--> 4 * 3 * 2 * 1 * 1
--> 24
```

- Each time `fact n` is called, that comes with a promise that the eventual value returned will be multiplied by `n`
- This "promise to do something with the return value of a function call" is stored in the control context, which is a control-flow feature of a processor that keeps track of the context in which functions are called. The control context of each call to fact is annotated above.
- Observe how above the control context grows with `n`: This can result in practical performance problems: if you've ever hit a "stack overflow" error in your code, it is typically because of accidental unbounded control context.

3 Tail Form

- There is an alternative way to write fact that does not grow the control context by using an **accumulator** `acc`:

```
let rec fact_acc (n:int) (acc : int) =
  if n = 0 then acc else (fact_acc (n - 1) (n * acc))

let fact_iter n = fact_iter_acc n 1
```

- Let's examine the control context when we evaluate this function:

```
fact_acc 4 1
--> fact_acc 3 2
--> fact_acc 2 12
--> fact_acc 1 24
--> fact_acc 0 24
--> 24
```

- Observe how instead of storing all the necessary data to finish the factorial computation in the control context, it is instead stored in the accumulator
- A function that requires no additional control context in order to execute exhibits **iterative control behavior**. A function that *does* require control context is said to exhibit **recursive control behavior**.
- How can you tell if a function has iterative or recursive control? If a function invokes itself in **operand position** (i.e., a recursive call is made as an argument to some other function), then it requires recursive control. Observe how in `fact`, the recursive call is invoked as an argument to the multiplication `*`.
- A **tail call** is a function call performed as the final act of a function.
- Observe that `fact_acc` above only ever calls itself as a tail call. We say `fact_acc` is in tail form.
- Tail calls enable a host of powerful compiler optimizations and are often essential for writing high-performance functional programs. Consequently, OCaml supports a tailcall annotation that raises a warning if a function is not invoked as a tail call:

```
let rec fact_iter_acc n acc =
  if n = 0 then acc else ((fact_iter_acc [@tailcall]) (n - 1) (n * acc))
```

4 Continuations

- In the fact example above, we are able to capture all the context necessary to finish the computation using an accumulator. This is not always possible; sometimes you need to keep track of more complicated behavior in the control context. This brings us to the notion of a continuation.
- Instead of using an accumulator, let's instead use a function to keep track of the remaining steps of computation. We will call this function a **continuation** (it "continues the computation")
- Let's switch over to Racket syntax to see this:

```
(define (fact_cont_h n k)
  (if (equal? n 0)
      (k 1) ; call continuation with 1
      (fact_cont_h
       (- n 1)
       (λ (r)
         ; call continuation with n * r
         (k (* n r))))))

(define (fact n) (fact_cont_h n (λ (r) r)))
```

- The function `fact_cont_h` is initially called with a continuation `(λ (r) r)`. We typically denote the argument to a continuation as `r`, short for "returned value". We initialize the continuation to the identity function, indicating that we have no further work to. Think of this as the "empty call stack".
- A function in tail form that takes a continuation as an argument is said to be in **continuation-passing form**. Function in continuation-passing form do not typically end by returning a value; instead, they end by calling the continuation.
- The best way to understand this code is to step through it and see how it avoids using control context (read this carefully! the scoping rules for the continuation `k` are very important):

```
fact_cont_h 2 (λ (r) r)                    ; label (λ (r) r) as id
--> fact_cont_h 1 (λ (r) (id (* 2 r)))    ; label (λ (r) (id (* 2 r))) as k1
--> fact_cont_h 0 (λ (r) (k1 (* 1 r)))
--> (λ (r) (k1 (* 1 r))) 1
--> (λ (r) (id (* 2 r))) 1
--> (id (* 2 1))
--> 2
```

- Notice how the continuation grows with each recursive call instead of the control context: it is keeping track of all the remaining computation to do once the recursive call finally hits its base case.
- Note the order in which the multiplications are performed: first the multiplication `1 * 1` is performed; then `2 * 1`.

5 The Tail-form Recipe

- Given a recursive function that is not in tail form, how can we transform it into one that is in tail form by using a continuation?
- Let's start with functions with a single argument that call themselves but no other functions.
- Let's look at the recursively-implemented factorial function again:

```
(define (fact-rec n)
  (if (equal? n 0) 1
      (* n (fact-rec (- n 1)))))
```

- Step 1: change the signature of the function to add a continuation:

```
(define (fact-iter n k)
  (if (equal? n 0) 1
      (* n (fact-iter (- n 1)))))
```

- Step 2: Find all points where the function returns a value and call the continuation at those points:

```
(define (fact-iter n k)
  (if (equal? n 0) (k n)
      (k (* n (fact-iter (- n 1))))))
```

- Step 3: Replace all instances where a function is called in operand form with a version of the function where it is called in tail form:

```
(define (fact-iter n k)
  (if (equal? n 0) (k n)
      (fact-iter (- n 1) ???))))
```

- Step 4: Move the operations that were previously dependent on the result of the recursive call inside the continuation, and replace the recursive call with the continuation's argument:

```
(define (fact-iter n k)
  (if (equal? n 0) (k n)
      (fact-iter (- n 1) (λ (r) (k (* n r)))))))
```

- If you don't want to remember this recipe, that's OK. If your function is in tail form, you are essentially required to write it this way.

6 Tail Form with Multiple Self-Calls

- Many recursive functions involve calling themselves multiple times in the same operand position.
- For example, the Fibonacci function looks like this:

```
(define (fib n)
  (match n
    [0 0]
    [1 1]
    [n (+ (fib (- n 1))
          (fib (- n 2)))]))
```

- How do we write this function in tail-form? We can follow a similar recipe, except this time, the continuation will be required to invoke `fib`.
- 1. Add a continuation:

```
(define (fib n k)
  (match n
    [0 0]
    [1 1]
    [n (+ (fib (- n 1))
          (fib (- n 2)))]))
```

- 2. Call continuation on return points:

```
(define (fib n k)
  (match n
    [0 (k 0)]
    [1 (k 1)]
    [n (k (+ (fib (- n 1))
          (fib (- n 2))))]))
```

- 3. Make operand calls tail-form. Here we have a choice about which operand call to make tail-form! We pick one of them arbitrarily:

```
(define (fib n k)
  (match n
    [0 (k 0)]
    [1 (k 1)]
    [n (fib (- n 1) ???))])
```

- 4. Fill in continuation:

```
(define (fib n k)
  (match n
    [0 (k 0)]
    [1 (k 1)]
    [n (fib (- n 1) (λ (r)
                      (k (+ r (fib (- n 2))))))]))
```

- Notice: we aren't done yet, since the above function is not in tail-form. The continuation itself has a call to `fib` that is not a tail-call. So, we simply repeat steps 3 and 4. Making the operand call tail-form, and filling in its continuation, yields a nested continuation:

```
(define (fib n k)
  (match n
    [0 (k 0)]
    [1 (k 1)]
    [n (fib (n - 1) (λ (r1)
                      (fib (- n 2) (λ (r2) (k (+ r1 r2))))))]))
```

7 Continuation-passing Interpreters

- Explicit representation of the continuation is a powerful tool for implementing control-flow operators in interpreters
- First, let's see how to implement an interpreter in continuation-passing form: this way, we have the control context explicitly available. Then, we will make use of this explicit control context in order to do interesting control flow operations.
- Recall the usual recursive calculator interpreter:

```
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)

(define (interp-recursive e)
  (match e
    [(enum n) n]
    [(eadd e1 e2) (+ (interp-recursive e1) (interp-recursive e2))]))
```

- Observe that this interpreter is *not* in tail-form. So, let's write this interpreter in tail-form by introducing a continuation and applying the tail-form recipe:

```
(define (interp-iter-h e k)
(match e
  [(enum n) (k n)]
  [(eadd e1 e2)
   (interp-iter-h e1
                (λ (r1)
                  (interp-iter-h e2 (λ (r2)
                                    (k (+ r1 r2))))))]))
```

- This interpreter is now in continuation-passing form.

8 The Power of Continuations: Implementing Return

- Continuations enable implementing interesting control-flow operations in our interpreters.
- Many programming languages have a `return` operation that immediately ends a function by returning a value.
- We can implement a return construct in our calculator language using continuations. The syntax of this new feature will be:

```
(struct enum (n) #:transparent)
(struct eadd (e1 e2) #:transparent)
(struct eret (v) #:transparent)
```

- As an example of the semantics of returning, the program `(eadd (enum 10) (eret 30))` should evaluate to 30.
- We can implement these semantics by *not calling the continuation*:

```
(define (interp-ret-h e k)
  (match e
    [(enum n) (k n)]
    [(eadd e1 e2)
     (interp-ret-h e1
                  (λ (r1)
                    (interp-ret-h e2 (λ (r2)
                                       (k (+ r1 r2))))))]
    [(eret v)
     ; drop the continuation (i.e., don't call it) and simply return v
     v]))
```