Control

Our Languages So Far

Our Languages So Far

What We Sometimes Need

What We Sometimes Need

- Escaping because of an error (exceptions)
- Escaping because we found the answer (early return)
- Revisiting an earlier decision we made (backtracking)
- Alternating between different computations (coroutines)

- These are all forms of ***control*** operations
  - I.e., of deviating from the normal control flow of our program

Control

- Control is all about deciding what to execute next
- May not be what directly follows in the program!

- **Our strategy:** make "what to execute next" explicit in our interpreter
    - Then implementing control operators is just a matter of messing with that

Continuation-passing style

Key idea: convert the interpreter into a style where all remaining work is explicit as an argument to the interpreter: a **continuation**

Kind of like what we did with `interp2` when we implemented state using a store: the `k` argument was a continuation!

Then we can swap in and out different pieces of work as we decide what we want to run!

Continuation-passing style

We will transform our interpreter from:

```
interp : FAE DefSub -> FAE-Value
```

into a function with this type:

```
FAE DefSub (FAE-Value -> FAE-Value)
-> FAE-Value
```

If we also have a store as a result, where does it go?

```
interp : (-> BFAE
             DefSub
             Store
             (Value*Store -> Value*Store)
             Value*Store)
```

(But we won't worry about stores for now.)

Analogy

If a store is akin to a **heap** as an explicit value...

...then a continuation is a **stack** as an explicit value!

What follows in the FAE interpreter, transformed in continuation-passing style. Each future step of computation is explicitly packaged up into a more complex `k` argument to be supplied to the next call to `interp`

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?)
       (rhs FAE?)]
  [sub (lhs FAE?)
       (rhs FAE?)]
  [id (name symbol?)]
  [fun (param-name symbol?)
       (body FAE?)]
  [app (fun-expr FAE?)
       (arg-expr FAE?)])
```

```
(define-type FAE-Value
  [numV (n number?)]
  [closureV (param-name symbol?)
            (body FAE?)
            (ds DefSub?)])

(define-type DefSub
  [mtSub]
  [aSub (name symbol?)
        (value FAE-Value?)
        (rest DefSub?)])
```

```
(define (interp-expr a-fae)
  (interp a-fae (mtSub)
          (λ (x) x)))
```

```
; FAE? DefSub? (FAE-Value? -> any) -> any
(define (interp a-fae ds k)
  (type-case FAE a-fae
    [num (n) (k (numV n))]
    [add (l r) (numop + l r ds k)]
    [sub (l r) (numop - l r ds k)]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
         (k (closureV param-name body ds))]
    [app (fun-expr arg-expr)
         the next slide contains this case]))
```

```
...
[app (fun-expr arg-expr)
     (interp fun-expr ds
       (λ (fun-val)
         (interp arg-expr ds
           (λ (arg-val)
             (interp
              (closureV-body fun-val)
              (aSub (closureV-param-name fun-val)
                    arg-val
                    (closureV-ds fun-val))
              k)))))]
```

```
(define (numop op l r ds k)
  (interp l ds
          (λ (l-v)
            (interp r ds
                    (λ (r-v)
                      (k (numV
                          (op (numV-n l-v)
                              (numV-n r-v)))))))))
```

```
(define (lookup name ds)
  (type-case DefSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (n num rest)
          (if (equal? n name)
              num
              (lookup name rest))]))
```

Let's add early return to our language!

To start, let's allow only 0 as an early return value

```
(define-type KFAE
  [num (n number?)]
  [add (lhs KFAE?)
       (rhs KFAE?)]
  [sub (lhs KFAE?)
       (rhs KFAE?)]
  [id  (name symbol?)]
  [fun (param-name symbol?)
       (body KFAE?)]
  [app (fun-expr KFAE?)
       (arg-expr KFAE?)]
  [ret-0]) ; no extra info to keep track of!
```

Ret-0

```
{{fun {x} {+ x {ret-0}}}
 5} ⇒ 0

{+ {{fun {x} {+ x {ret-0}}}
    5}
   3}
⇒ 3

{ret-0} ⇒ error: not inside a function
```

Ret-0

```
...
[ret-0 () (numV 0)]
```

- We don't *have* to call our continuation.
- If we ignore it, we skip its work!

Ret-0

```
{+ {{fun {x} {+ x {ret-0}}}
    5}
   3}
⇒ 0
```

- Oops, we return too far!
- All the way to the beginning, in fact!

- Solution: two continuations! One for normal execution, one for returning!

```
(define (interp-expr a-kfae)
  (interp a-kfae (mtSub)
          (λ (x) x)
          (λ (x)
            (error 'interp
                   "not inside a function"))))
```

If we produce a value, continue interpreting the current function.

```
; KFAE? DefSub?
;   (KFAE-Value? -> KFAE-Value?)
;   (KFAE-Value? -> KFAE-Value?)
;    -> KFAE-Value?
(define (interp a-kfae ds k ret-k)
  (type-case KFAE a-kfae
    [num (n) (k (numV n))]
    [add (l r) (numop + l r ds k ret-k)]
    [sub (l r) (numop - l r ds k ret-k)]
    [id (name) (k (lookup name ds))]
    [fun (param-name body)
         (k (closureV param-name body ds))]
    ...))
```

```
...
[app (fun-expr arg-expr)
     (interp fun-expr ds
       (λ (fun-val)
         (interp arg-expr ds
           (λ (arg-val)
             (interp
              (closureV-body fun-val)
              (aSub (closureV-param-name fun-val)
                    arg-val
                    (closureV-ds fun-val))
              k
              ; we're entering a new function body
              ; if we return from it, it's as if we
              ; were done interpreting the body!
              ; so we're done with the call!
              k))
           ret-k))
       ret-k)]
```

Returning = calling the return continuation with the return value!

```
...
[ret () (ret-k (numV 0))]
```

For completeness

```
(define (numop op l r ds k ret-k)
  (interp l ds
          (lambda (l-v)
            (interp r ds
                    (lambda (r-v)
                      (k (numV
                          (op (numV-n l-v)
                              (numV-n r-v)))))
                    ret-k))
          ret-k))
```

Pass `ret-k` along in case either operand returns.

Otherwise continue execution as normal

Returning any value

Let's generalize to allow any return value

```
(define-type KFAE
  [num (n number?)]
  [add (lhs KFAE?)
       (rhs KFAE?)]
  [sub (lhs KFAE?)
       (rhs KFAE?)]
  [id  (name symbol?)]
  [fun (param-name symbol?)
       (body KFAE?)]
  [app (fun-expr KFAE?)
       (arg-expr KFAE?)]
  [ret-0]
  [ret (ret-expr KFAE?)])
```

Returning any value

```
{{fun {x} {+ x {ret 2}}}
 5} ⇒ 2

{+ {{fun {x} {+ x {ret 10}}}
    5}
   3}
⇒ 13

{ret 2} ⇒ error: not inside a function
```

```
...
[ret (ret-expr)
     ; compute your return value
     (interp ret-expr ds
             ; when you're done, return!
             (lambda (ret-val) (ret-k ret-val))
             ; if someone tries to return while
             ; computing the return value, that's
             ; the same as just returning
             ret-k)]
```

...which is equivalent to

```
...
[ret (ret-expr)
     (interp ret-expr ds
             ret-k ; that lambda was extraneous
             ret-k)]
```

Ret within Ret

`ret` is an expression

So can have `ret` inside `ret`!

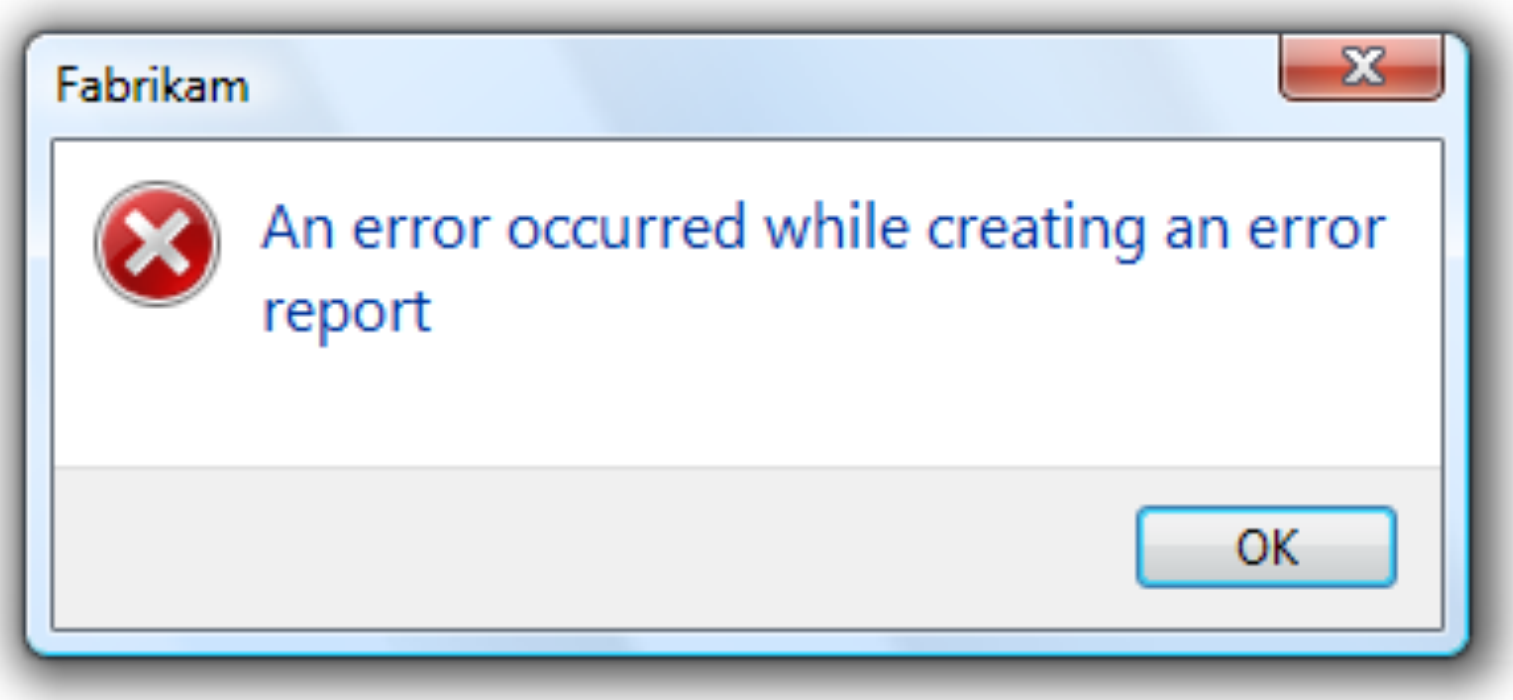
```
{{fun {x} {ret {ret 2}}}
 5} ⇒ 2

{{fun {x} {+ x {ret {+ 4 {ret 2}}}}}
 5}
⇒ 2
```

That's a bit weird, but it follows naturally from our rules.

This kind of behavior makes sense for, e.g., exceptions.

Exception within Exception



Source: https://docs.microsoft.com/en-us/windows/desktop/uxguide/mess-error