

Exceptions and Continuations

David Bremner

March 19, 2025

Exception handling

```
(define exception (make-parameter identity))
```

```
(define (throw msg)
  ((parameter-ref exception) msg))
```

```
(define (try thunk recovery)
  (let/cc esc
    (parameterize ([exception
                     (lambda (x) (esc (recovery x)))]])
      (thunk))))
```

Using the exception handler

```
throw (try  
  (lambda ()  
    (begin  
      (throw "abort!")  
      (/ 1 0)  
      (display "done"))))  
  (lambda (x)  
    (display (list "caught" x)))))
```

Adding abstract syntax for error

```
(define-type Exp
  [numE (n : Number)]
  [plusE (lhs : Exp) (rhs : Exp)]
  [minusE (lhs : Exp) (rhs : Exp)]
  [varE (name : Symbol)]
  [lamE (param : Symbol) (body : Exp)]
  [appE (fun-expr : Exp) (arg-expr : Exp)]
  [errorE (msg : String)] ;; New
  [if0E (test : Exp) (then : Exp) (else : Exp)])
```

Add an error value

```
(define-type Value  
  [numV (n : Number)]  
  [errorV (msg : String)]  
  [funV (param : Symbol)  
        (body : Exp)  
        (env : Env)])
```

A first attempt, errors as values

```
(define (interp expr env)
  (type-case Exp expr
    :
    [(errorE msg) (errorV msg)]
    [(lamE bound-id bound-body)
     (funV bound-id bound-body env)]
    [(appE fun-expr arg-expr)
     (let ([fval (interp fun-expr env)])
       (type-case Value fval
         [(funV bound-id bound-body f-env)
          (interp bound-body
                   (Extend bound-id (interp arg-expr env)
                           f-env))]
         [else (error 'eval "not a function")]))]))
```

Error values need special treatment

```
rec3 (test (run `{let1 f {lam {x} {+ x 1}}  
              {+ {f 2} {error "abort!"}}}))  
      (errorV "abort!"))
```

Approach I: error propagation

```
(define (arith-op op val1 val2)
  (cond
    [(and (numV? val1) (numV? val2))
     (numV (op (numV-n val1)
               (numV-n val2)))]
    [(errorV? val1) val1]
    [(errorV? val2) val2]
    [else (error 'arith-op "got function")]))
```

Good news, bad news.

rec4

```
(test (run `{let1 f {lam {x} {+ x 1}}  
              {+ {f 2} {error "abort!"}}}))  
(errorV "abort!"))
```

```
(test (run `{let1 f {lam {x} {+ x 1}}  
              {+ {f {error "abort!"}} 2}}))  
(errorV "abort!"))
```

```
(test (run `{if0 {error "abort!"} 1 2})  
(errorV "abort!"))
```

What's the best approach?

- ▶ we could find all the code paths and add error value handling code
- ▶ we could use `plait` exceptions to implement `interp` exceptions (`meta`-interpreter pros and cons)
- ▶ we could use continuation passing style to implement more flexible control flow

Syntactic continuations

```
(define (interp expr env)
  (type-case Exp expr
    [(plusE l r)
     (arith-op
      +
      (interp l env)
      (interp r env))])
  :)
```

```
(define (interp expr env k)
  (type-case Exp expr
    [(plusE l r)
     (interp l env
              (plusSecondK
               r env k))])
  :)
```

Replace return with continue

```
(define (interp expr env)
  (type-case Exp expr
    [(numE n) (numV n)]
    :))
```

```
(define (interp expr env k)
  (type-case Exp expr
    [(numE n)
     (continue k (numV n))]
    :))
```

- ▶ continue interprets **syntactic** continuations, rather than using plait functions/continuations.

Functions and Identifiers are like numbers

```
(define (interp expr env k)
  (type-case Exp expr
    [(numE n) (continue k (numV n))]
    [(varE name) (continue k (lookup name env))]
    [(lamE param body-expr)
     (continue k (funV param body-expr env))]
    :)))
```

previously return a value

now pass a value to the continuation.

Interpreting continuations

```
(define (continue [k : Continuation] [v : Value]) :  
  Value  
  (type-case Continuation k  
    [(emptyCont) v]  
    [(plusSecondK r env next-k)  
     (interp r env (doPlusK v next-k))]  
    [(doPlusK v1 next-k)  
     (continue next-k (num+ v1 v))]  
    :))
```

- ▶ roughly, interp converts to CPS, and continue interprets that.
- ▶ continue, interp are tail recursive.
- ▶ call stack replaced by continuation data structure k

Simple evaluation

```
trace (trace interp)
      (trace continue)
      (run `{+ 1 4})
      ;(run `{- 7 {+ 1 4}})
```

Calling functions

```
trace (trace interp)
      (trace continue)
      (run `{{lam {x} {+ 1 x}} 4})
      ;(run `{{let1 f {lam {x} {+ 1 x}} {f 4}}})
```

Exceptions

- ▶ The case for `errorE (throw)` looks the same as before
- ▶ ignores current continuation

```
[(errorE msg) (errorV msg)]
```

`trace`

```
(trace interp)
(run `{\let1 f {\lam {x} {+ x 1}}
        {+ {f 2} {error "abort!"}}})
```

```
#;(run `{\let1 f {\lam {x} {+ x 1}}
        {+ {f {error "abort!"}} 2}})
```

```
#;(run `{\if0 {error "abort!"} 1 2})
```