Store-Passing Interpreters

Our **`BCFAE`** interpreter explains state by representing the store as a value

- Every step in computation produces a new store
- The interpreter itself is purely functional

It's a ***store-passing interpreter***

Variables

Boxes don’t explain one of our earlier Scheme examples:

```
(define counter 0)
(define (f x)
  (begin
    (set! counter (+ x counter))
    counter))
```

In a program like this, an identifier no longer stands for a ***value***; instead, an identifier stands for a ***variable***

Implementing Variables

Option 1:

```
(define counter 0)
(define (f x)
  (begin
    (set! counter (+ x counter))
    counter))
(f 10)

⇒   (define counter (box 0))
    (define (f x)
      (begin
        (set-box! counter (+ (unbox x)
                             (unbox counter)))
        (unbox counter)))
    (f (box 10))
```

Option 2:

- Essentially the same, but hide the boxes in the interpreter

BMCFAE = BCFAE + variables

```
<BMCFAE>  ::=  <num>
            |  {+ <BMCFAE> <BMCFAE>}
            |  {- <BMCFAE> <BMCFAE>}
            |  <id>
            |  {fun {<id>} <BMCFAE>}
            |  {<BMCFAE> <BMCFAE>}
            |  {if0 <BMCFAE> <BMCFAE> <BMCFAE>}
            |  {newbox <BMCFAE>}
            |  {setbox <BMCFAE> <BMCFAE>}
            |  {openbox <BMCFAE>}
            |  {seqn <BMCFAE> <BMCFAE>}
            |  {set <id> <BMCFAE>}                  NEW
```

Implementing Variables

```
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?)
        (address integer?)
        (ds DefrdSub?)])
```

Implementing Variables

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [id (name) (v*s (store-lookup (lookup name ds) st)
                   st)]
  ...)
```

Implementing Variables

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [app (fun-expr arg-expr)
       (interp-two fun-expr arg-expr ds st
                   (lambda (fun-val arg-val st)
                     (local [(define a (malloc st))]
                       (interp (closureV-body fun-val)
                               (aSub (closureV-param fun-val)
                                     a
                                     (closureV-sc fun-val))
                               (aSto a
                                     arg-val
                                     st)))))]
  ...)
```

Implementing Variables

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [set (id val-expr)
       (local [(define a (lookup id ds))]
         (type-case Store*Value (interp val-expr ds st)
           [v*s (val st)
                (v*s val
                     (aSto a
                           val
                           st))]))]
  ...)
```

Variables and Function Calls

```
(define (swap x y)
  (local [(define z y)]
    (set! y x)
    (set! x z)))

(local [(define a 10)
        (define b 20)]
  (begin
    (swap a b)
    a))
```

Result is `10`; assignment in `swap` cannot affect `a`

Call-by-Reference

What if we wanted **swap** to change **a**?

```
(define (swap x y)       ⇒    (define (swap x y)
  (local [(define z y)]         (local [(define z (box (unbox y)))]
    (set! y x)                    (set-box! y (unbox x))
    (set! x z)))                  (set-box! x (unbox z))))

(local [(define a 10)         (local [(define a (box 10))
        (define b 20)]                (define b (box 20))]
  (begin                        (begin
    (swap a b)                    ; (swap (box (unbox a))
    a))                           ;       (box (unbox b)))
                                  (swap a b)
                                  (unbox a)))
```

This is called ***call-by-reference***, as opposed to ***call-by-value***

Terminology alert: this “call-by-value” is orthogonal to the use in “call-by-value” vs. “call-by-name”

Implementing Call-by-Reference

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [app (fun-expr arg-expr)
       (if (id? arg-expr)
           ; call-by-ref handling for id arg:
           (type-case Value*Store (interp fun-expr ds st)
             [v*s (fun-val st)
                  (local [(define a
                            (lookup (id-name arg-expr) ds))]
                    (interp (closureV-body fun-val)
                            (aSub name
                                  a
                                  (closureV-sc fun-val))
                            st))])
           ; as before:
           ...)]
  ...)
```