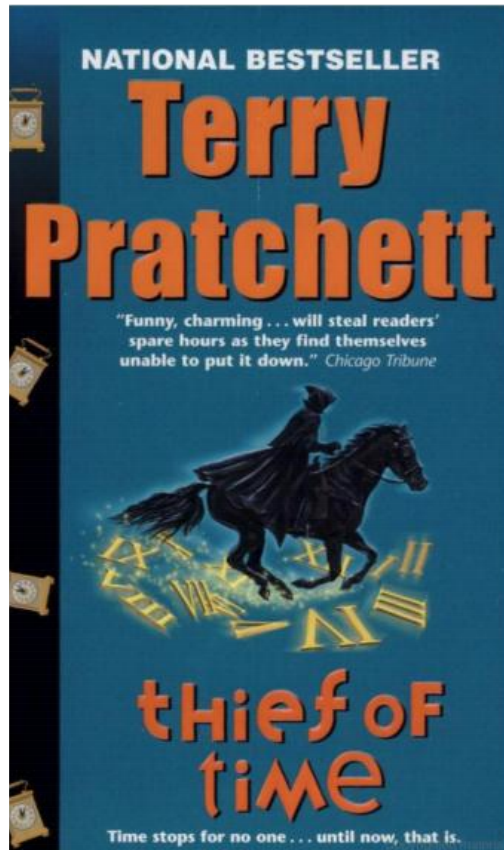


Thief of Time



According to the First Scroll of Wen the Eternally Surprised, Wen stepped out of the cave where he had received enlightenment and into the dawning light of the first day of the rest of his life. He stared at the rising sun for some time, because he had never seen it before.

He prodded with a sandal the dozing form of Clodpool the Apprentice, and said: "I have seen. Now I understand."

Then he stopped and looked at the thing next to Clodpool. "What is that amazing thing?" he said.

"Er . . . er . . . it's a tree, master," said Clodpool, still not quite awake. "Remember? It was there yesterday."

"There was no yesterday."

"Er . . . er . . . I think there *was*, master," said Clodpool, struggling to his feet. "Remember? We came up here, and I cooked a meal, and had the rind off your *sklang* because you didn't want it."

"I *remember* yesterday," said Wen, thoughtfully. "But the memory is in my head *now*. Was yesterday real? Or is it only the memory that is real? Truly, yesterday I was not born."

Clodpool's face became a mask of agonized incomprehension.

"Dear stupid Clodpool, I have learned everything," said Wen. "In the cup of the hand there is no past, no future. There is only now. There is no time but the present. We have a great deal to do."

Wen the Eternally Surprised

The first question they ask is: “Why was he eternally surprised?”

And they are told: “Wen considered the nature of time and understood that the universe is, instant by instant, re-created anew. Therefore, he understood, there is, in truth, no Past, only a memory of the Past. Blink your eyes, and the world you see next did not exist when you closed them. Therefore, he said, the only appropriate state of the mind is surprise. The only appropriate state of the heart is joy. The sky you see now, you have never seen before. The perfect moment is now. Be glad of it.”

Part I

Mutable Structures

Functional Programs

So far, the language that we've implemented is purely ***functional***:

A function produces the same result every time for the same arguments





Non-Functional Procedures

```
(define (f x)
  (+ x (read)))
```

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

```
(define f
  (local [(define b (box 0))]
    (lambda (x)
      (begin
        (set-box! b (+ x (unbox b)))
        (unbox b))))))
```

BCFAE = FAE + Boxes

```
<BCFAE> ::= <num>
| {+ <BCFAE> <BCFAE>}
| {- <BCFAE> <BCFAE>}
| <id>
| {fun {<id>} <BCFAE>}
| {<BCFAE> <BCFAE>}
| {newbox <BCFAE>} 
| {setbox <BCFAE> <BCFAE>} 
| {openbox <BCFAE>} 
| {seqn <BCFAE> <BCFAE>} 
```

```
{with {b {newbox 0}}
  {seqn
    {setbox b 10}
    {openbox b}}}} ⇒ 10
```

Implementing Boxes with Boxes

```
(define-type BCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body BCFAE?)
            (ds DefrdSub?)]
  [boxV (container (box-of BCFAE?))])
```

Implementing Boxes with Boxes

```
; interp : BCFAE DefrdSub -> BCFAE-Value
(define (interp a-bcfae ds)
  (type-case RCFAE a-bcfae
    ...
    [newbox (val-expr)
            (boxV (box (interp val-expr ds)))]
    [setbox (box-expr val-expr)
            (set-box! (boxV-container
                      (interp box-expr ds))
                      (interp val-expr ds))]
    [openbox (box-expr)
             (unbox (boxV-container
                    (interp box-expr ds)))]))
```

But this doesn't explain anything about boxes!

Boxes and Memory

```
{with {b {newbox 7}}  
  ...}
```

 ⇒ ...

Memory:

Memory:

			7	

Boxes and Memory

... {setbox b 10}

...

Memory:

			7	

⇒

... {openbox b}

...

Memory:

			10	

The Store

We represent memory with a **store**:

```
(define-type Store
  [mtSto]
  [aSto (address integer?)
        (value BCFAE-Value?)
        (rest Store?) ])
```

Memory:

			10	

```
(aSto 13 (numV 10)
      (mtSto))
```

Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
```

```
(define-type BCFAE-Value  
  [numV (n number?)]  
  [closureV (param symbol?)  
            (body BCFAE?)  
            (ds DefrdSub?)]  
  [boxV (address integer?)])
```

```
(define-type Value*Store  
  [v*s (value BCFAE-Value?)  
       (store Store?)])
```

Implementing Boxes without State

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

Implementing Boxes without State

```
; malloc : Store -> integer
(define (malloc st)
  (+ 1 (max-address st)))

; max-address : Store -> integer
(define (max-address st)
  (type-case Store st
    [mtSto () 0]
    [aSto (n v st)
           (max n (max-address st))]))
```

Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [openbox (bx-expr)
    (type-case Value*Store (interp bx-expr ds st)
      [v*s (bx-val st)
        (v*s (store-lookup (boxV-address bx-val)
                           st)
              st)]]])
  ...)
```

Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [setbox (bx-expr val-expr)
    (type-case Value*Store (interp bx-expr ds st)
      [v*s (bx-val st2)
        (type-case Value*Store (interp val-expr ds st2)
          [v*s (val st3)
            (v*s val
              (aSto (boxV-address bx-val)
                    val
                    st3))])])])
  ...)
```

seqn, **add**, **sub**, and **app** will need the same sort of sequencing

Implementing Boxes without State

```
; interp-two : (BCFAE BCFAE DefrdSub Store
;             (Value Value Store -> Value*Store)
;             -> Value*Store)
(define (interp-two expr1 expr2 ds st handle)
  (type-case Value*Store (interp expr1 ds st)
    [v*s (val1 st2)
      (type-case Value*Store (interp expr2 ds st2)
        [v*s (val2 st3)
          (handle val1 val2 st3)]))]))
```

Implementing Boxes without State

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [add (l r) (interp-two l r ds st
                        (lambda (v1 v2 st)
                          (v*s (num+ v1 v2) st)))]
  ...
  [seqn (a b) (interp-two a b ds st
                          (lambda (v1 v2 st)
                            (v*s v2 st)))]
  ...
  [setbox (bx-expr val-expr)
          (interp-two bx-expr val-expr ds st
                    (lambda (bx-val val st3)
                      (v*s val
                          (aSto (boxV-address bx-val)
                                val
                                st3)))))]
  ...)
```

Store-Passing Interpreters

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

- Every step in computation produces a new store

the universe is, instant by instant, re-created anew

- The interpreter itself is purely functional

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

Part II

Mutable Variables

Variables

Boxes don't explain one of our earlier Racket 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>}
```



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-ds fun-val))
            (aSto a
              arg-val
              st))))))]
  ...)
```

Implementing Variables

```
; interp : BCFAE DefrdSub Store -> Value*Store
(define (interp expr ds st)
  ...
  [setid (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-ds fun-val))
                          st)))]
      ; as before:
      ...)]
  ...)
```