CS4613 Lecture 10: Safety

David Bremner

October 8, 2025

Calculator Revisited

p. 133

calc

```
(calc : (Exp -> Value))
(define (calc e)
  (type-case Exp e
    [(num n) (numV n)]
    [(str s) (strV s)]
    [(plus l r) (num+ (calc l) (calc r))]
    [(cat l r) (str++ (calc l) (calc r))]))
```

Safety Checks

p. 134

calc

```
(define (str++ lv rv)
  (type-case Value lv
    ((strV ls)
     (type-case Value rv
       ((strV rs) (strV (string-append ls rs)))
       (else (error '++ "right not a string"))))
    (else (error '++ "left not a string"))))
```

- num+ is similar

Checking the checks

calc

```
(test (calc (plus (num 1) (num 2))) (numV 3))
(test (calc (plus (num 1) (plus (num 2) (num 3))))
   (numV 6))
(test (calc (cat (str "hel") (str "lo")))
      (strV "hello"))
(test (calc (cat (cat (str "hel") (str "l"))
                 (str "o"))) (strV "hello"))
(test/exn (calc (cat (num 1) (str "hello"))) "left")
(test/exn (calc (plus (num 1) (str "hello"))) "right")
```

- this is essentially the same as the "Dynamic type checking" example from Lecture 8.

Simulating explicit memory allocation

memcalc1

```
(define MEMORY (make-vector 100 -1))
(define next-addr 0)
(define (reset) (set! next-addr 0))
(define (write-and-bump v)
  (let ([n next-addr])
    (begin
      (vector-set! MEMORY n v)
      (set! next-addr (add1 next-addr))
      n)))
```

Reading and Writing numbers

```
(define (store-num n)
  (write-and-bump n))
(define (read-num a)
  (vector-ref MEMORY a))
```

memcalc1

```
(test (read-num (store-num 42)) 42)
```

Writing strings

p. 137

```
(define (store-str s)
  (let ([a0 (write-and-bump (string-length s))])
    (begin
      (map write-and-bump
           (map char->integer (string->list s)))
      a0)))
```

memcalc2

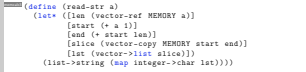
```
(test (begin (reset) (store-str "hello")
             (vector-copy MEMORY 0 6))
      '#(5 104 101 108 108 111))
```

Reading strings back

memcalc3

```
(define (read-str a)
  (let* ([len (vector-ref MEMORY a)]
         [start (+ a 1)]
         [end (+ start len)]
         [slice (vector-copy MEMORY start end)]
         [lst (vector->list slice)])
    (list->string (map integer->char lst))))
```

Reading strings back



```
(define (read-str a)
  (let* ([len (vector-ref MEMORY a)]
         [start (+ a 1)]
         [end (+ start len)]
         [slice (vector-copy MEMORY start end)]
         [lst (vector->list slice)])
    (list->string (map integer->char lst))))
```

1. This is simplified compared to the book by using the imported functions `vector-copy` and `vector->list`. For a more efficient (and low level) version, see the book

Testing strings

memcalc3

```
(test (read-str (store-str "cookie monster"))
      "cookie monster")
```

Replacing types with “addresses”

p. 138

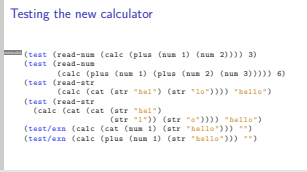
```
(define-type-alias Value Number)
(define numV store-num)
(define strV store-str)

(define (num+ la ra)
  (numV (+ (read-num la) (read-num ra))))
(define (str++ la ra)
  (strV (string-append (read-str la)
                       (read-str ra))))
```

Testing the new calculator

memcalc4

```
(test (read-num (calc (plus (num 1) (num 2)))) 3)
(test (read-num
       (calc (plus (num 1) (plus (num 2) (num 3))))) 6)
(test (read-str
       (calc (cat (str "hel") (str "lo")))) "hello")
(test (read-str
  (calc (cat (cat (str "hel")
                  (str "l")) (str "o")))) "hello")
(test/exn (calc (cat (num 1) (str "hello"))) "")
(test/exn (calc (plus (num 1) (str "hello"))) "")
```



1. We diverge from the book here in matching any error message; this so the tests pass if/when the error messages change

There are no errors. Everything is fine.

memcalc5

```
(read-num (calc (str "hello")))
;(read-str (calc (num 1)))
(read-num (calc (plus (num 1) (str "hello"))))
;(read-str (calc (plus (num 1) (str "hello"))))
```

Tagging Data

```
(define NUMBER-TAG 1337)
(define STRING-TAG 5712)

(define (store-num n)
  (let ([a0 (write-and-bump NUMBER-TAG)])
    (begin
      (write-and-bump n)
      a0)))
```

memcalc6

```
(test (begin (reset) (store-num 42)
             (vector-copy MEMORY 0 2))
      '#(1337 42))
```

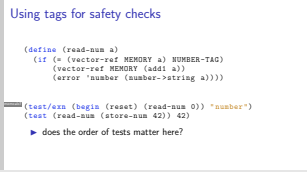
Using tags for safety checks

```
(define (read-num a)
  (if (= (vector-ref MEMORY a) NUMBER-TAG)
      (vector-ref MEMORY (add1 a))
      (error 'number (number->string a))))
```

memcalc7

```
(test/exn (begin (reset) (read-num 0)) "number")
(test (read-num (store-num 42)) 42)
```

- does the order of tests matter here?



1. We diverge from the book here in keeping the same name for tagged function, called `safe-read-num` in the book

Tagging strings

p. 141

```
(define (store-str s)
  (let ([a0 (write-and-bump STRING-TAG)])
    (begin
      (write-and-bump (string-length s))
      (map write-and-bump
           (map char->integer (string->list s)))
      a0)))
```

memcalc8

```
(test (begin (reset) (store-str "hello")
             (vector-copy MEMORY 0 7))
      '#(5712 5 104 101 108 108 111))
```

Reading tagged strings

```
(define (read-str a)
  (if (= (vector-ref MEMORY a) STRING-TAG)
      (let* ([len (vector-ref MEMORY (+ a 1))]
             [start (+ a 2)]
             [end (+ start len)]
             [slice (vector-copy MEMORY start end)]
             [lst (vector->list slice)])
        (list->string (map integer->char lst)))
      (error 'string (number->string a))))
```

memcalc9

```
(test (read-str (store-str "hello-world"))
   "hello-world")
(test/exn (read-str (store-num 42)) "string")
```

Using our tagged heap

p. 141

memcalcA

```
(define (num+ la ra)
  (store-num (+ (read-num la) (read-num ra))))
(define (str++ la ra)
  (store-str (string-append (read-str la)
                            (read-str ra))))
```

- we diverged from the text in re-using the names `read-num` and `read-str`, so no change is needed.

does not typecheck

p. 142

gread1
```
(define (generic-read a)
  (let ([tag (vector-ref MEMORY a)])
    (cond
      [(= tag NUMBER-TAG) (read-num a)]
      [(= tag STRING-TAG) (read-str a)]
      [else (error 'generic-read "invalid tag")])))
```

- why is this not valid `plait`?

Stringly typed version

p. 142

gread2

```
(define (generic-read a)
  (let ([tag (vector-ref MEMORY a)])
    (cond
      [(= tag NUMBER-TAG)
       (number->string (read-num a))]
      [(= tag STRING-TAG) (read-str a)]
      [else (error 'generic-read "invalid tag")])))
```

Using generic-read for tests

gread2

```
(define (run exp)
  (generic-read (calc exp)))

(test (run (plus (num 1) (num 2))) "3")
(test (run (plus (num 1) (plus (num 2) (num 3)))) "6")
(test (run (cat (str "hel") (str "lo"))) "hello")
(test (run (cat (cat (str "hel") (str "l")) (str "o")))
      "hello")
```

- what are the pros and cons of this approach?

Strongly typed generic-read

gread3

```
(define-type result-type
  [numR (n : Number)]
  [strR (s : String)])

(define (generic-read a)
  (let ([tag (vector-ref MEMORY a)])
    (cond
      [(= tag NUMBER-TAG) (numR (read-num a))]
      [(= tag STRING-TAG) (strR (read-str a))]
      [else (error 'generic-read "invalid tag")])))
```

- both are useful only at “the top level”

Using strongly typed `generic-read` for tests

gread3

```
(define (run exp)
  (generic-read (calc exp)))

(test (run (plus (num 1) (num 2))) (numR 3))
(test (run (plus (num 1) (plus (num 2) (num 3))))
   (numR 6))
(test (run (cat (str "hel") (str "lo"))) (strR
   "hello"))
(test (run (cat (cat (str "hel") (str "l")) (str "o")))
      (strR "hello"))
```

- what are the pros and cons of this approach?