

CS4613 Lecture 11: Safety and Soundness.

David Bremner

February 21, 2024

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?

A by-now standard typechecker

p. 141

tc-calc

```
(define (expect2 type a b)
  (if (and (equal? type (typecheck a))
          (equal? type (typecheck b))) type
      (error 'typecheck
              (string-append "expected 2 x "
                              (to-string type)))))
```

```
(define (typecheck exp)
  (type-case Exp exp
    [(num n) (numT)]
    [(str s) (stringT)]
    [(cat a b) (expect2 (stringT) a b)]
    [(plus a b) (expect2 (numT) a b)]))
```

Check then eval

p. 141

tc-calc

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

(define (run exp)
  (let* ([type (typecheck exp)]
        [loc (calc exp)])
    (type-case Type type
      [(numT) (numR (read-num loc))]
      [(stringT) (strR (read-str loc))])))
```


Typed tests

```
tc-calc (test (typecheck (plus (num 1) (num 2))) (numT))
(test (run (plus (num 1) (plus (num 2) (num 3))))
      (numR 6))
(test (run (cat (str "hel") (str "lo")))) (strR
      "hello"))
(test (typecheck
      (cat (cat (str "hel")
                (str "l")) (str "o"))) (stringT))
(test/exn (run (cat (num 1) (str "hello"))) "expected")
(test/exn (run (plus (num 1) (str "hello")))
      "expected")
```

What do we want?

notation

p. 142

Rule	Code
$\vdash e : T$	<code>(equal? (typecheck e) T)</code>
$e \rightarrow v$	<code>(equal? (interp e) v)</code>

In a perfect world

$$\vdash e : T \Leftrightarrow (e \rightarrow v) \wedge (v : T)$$

Soundness

Are you going to use all of those bits?

p. 144

- ▶ In general pointers to memory should be **aligned** to e.g. 4 byte or 8 byte boundaries
- ▶ This means the lower bits are ours to play with, if we are careful.

numtag

```
(define (ref->tag ref)
  (modulo ref 4))

(define (ref->word loc)
  (quotient loc 4))

(define (tag->word word tag)
  (+ tag (* 4 word)))
```

Testing the new API

```
numtag (let ([str-ref (tag-word 0 STRING-TAG)]  
            [num-ref (tag-word 7 NUMBER-TAG)])  
  (begin  
    (test (ref->tag str-ref) STRING-TAG)  
    (test (ref->word str-ref) 0)  
    (test (ref->tag num-ref) NUMBER-TAG)  
    (test (ref->word num-ref) 7)))
```

Creating tagged references

nt-store

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

```
(define (store-num n)
  (tag-word (write-and-bump n) NUMBER-TAG))
```

Testing tagged references

nt-store

```
(let ([str-ref (store-str "hello")]  
      [num-ref (store-num 42)])  
  (begin  
    (reset)  
    (test (ref->tag str-ref) STRING-TAG)  
    (test (ref->tag num-ref) NUMBER-TAG)  
    (test (vector-copy MEMORY (ref->word str-ref) 7)  
          '#(5 104 101 108 108 111 42))  
  ))
```

reading tagged references (numbers)

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

Reading tagged references (strings)

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


Putting it all together

```
tmemcalc (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"))) "")
```