

# Mutable Store

**15-411/15-611 Compiler Design**

Seth Copen Goldstein

March 11, 2025

# Today

- Pointers
- The Heap and pointers
- Arrays
- Length & bounds checking
- Elaboration of +=, etc.

# Adding a pointer

- Extend types

$$\tau ::= \text{int} \mid \text{bool} \mid \tau^*$$

- Extend expressions


– **alloc( $\tau$ )**: allocate a heap cell to hold a value of  $\tau$

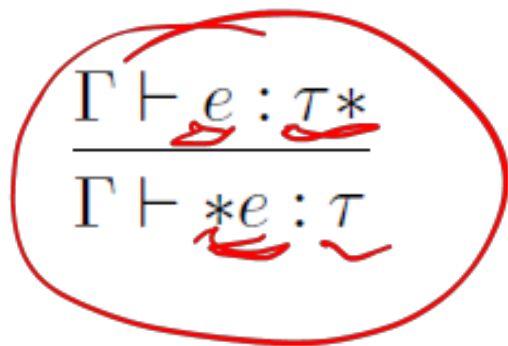
– **\* $e$** : dereference a pointer to get value at  $e$


– **null**: special null pointer

$$e ::= \dots \mid \text{alloc}(\tau) \mid *e \mid \text{null}$$

# Typing rules

$$\frac{}{\Gamma \vdash \text{alloc}(\tau) : \tau^*}$$


$$\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash *e : \tau}$$


$$\frac{}{\Gamma \vdash \text{null} : ?}$$


- A freshly allocated cell has type “pointer to  $\tau$ ”
- if  $e$  has type “pointer to  $\tau$ ,” then  $*e$  has type “ $\tau$ ”
- What type should null have?

not quite

# The type of null?

- Desired behavior
  - allow any pointer to be compared to null
  - disallow pointer dereference of null

*\*null*

*int \* p*

# Equality for pointers?

*int \* p = null;  
int \* q = p;*

- Can we compare  $\tau^*$  and  $\sigma^*$ :
  - if  $\tau = \sigma$  :
  - if  $\tau \neq \sigma$  :
  - What about `int* p; ... if (p==null) ...`
- null is given type of “any\*”
- And, implicitly converted to  $\tau^*$  as needed

# The type of null?

- Desired behavior
  - allow any pointer to be compared to null
  - disallow pointer dereference to null
- Using the type “any\*” along with subsumption
- Subsumption used for implicit coercion

$$\frac{}{\Gamma \vdash \text{null} : \text{any}^*} \qquad \frac{\Gamma \vdash e : \text{any}^*}{\Gamma \vdash e : \tau^*}$$

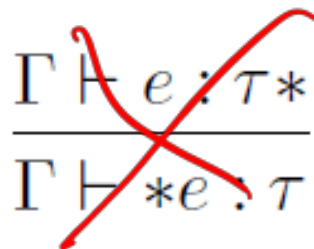
- Have to make sure introducing any\* is safe

# The type of null?

- Desired behavior
  - allow any pointer to be compared to null
  - disallow pointer dereference to null
- Using the type “any\*” along with subsumption
- Subsumption used for implicit coercion

$$\frac{}{\Gamma \vdash \text{null} : \text{any}^*} \qquad \frac{\Gamma \vdash e : \text{any}^*}{\Gamma \vdash e : \tau^*}$$

- Have to make sure introducing any\* is safe


$$\frac{\Gamma \vdash e : \tau^*}{\Gamma \vdash *e : \tau}$$



# The type of null?

- Desired behavior
  - allow any pointer to be compared to null
  - disallow pointer dereference to null
- Using the type “any\*” along with subsumption
- Subsumption used for implicit coercion

$$\frac{}{\Gamma \vdash \text{null} : \text{any}^*} \qquad \frac{\Gamma \vdash e : \text{any}^*}{\Gamma \vdash e : \tau^*}$$

- Can't allow **\*null**

$$\frac{\Gamma \vdash e : \tau^* \quad \Gamma \not\vdash e : \text{any}^*}{\Gamma \vdash *e : \tau}$$

# Typing rules (revised)

$$\frac{}{\Gamma \vdash \text{alloc}(\tau) : \tau^*}$$

$$\frac{\Gamma \vdash e : \tau^* \quad \Gamma \not\vdash e : \text{any}^*}{\Gamma \vdash *e : \tau}$$

$$\frac{}{\Gamma \vdash \text{null} : \text{any}^*}$$

$$\frac{\Gamma \vdash e : \text{any}^*}{\Gamma \vdash e : \tau^*}$$

- A freshly allocated cell has type “pointer to  $\tau$ ”
- if  $e$  has type “pointer to  $\tau$ ,” and  $e$  isn’t null, then  $*e$  has type “ $\tau$ ”
- null has the indefinite type
- Implicit coercion

# Representing the Heap

Evaluation of expression  $e$  in the context of

- a **Heap**,
- **Stack**, and
- binding **environment**.



$$\underline{H; S; \eta \vdash e \triangleright K}$$

- $\text{alloc}(\tau)$  returns an unused address in  $H$  (the heap) which can store a value of  $\tau$

# What is an address?

- How do we represent addresses, i.e., the result of the **alloc** operation?
- 64-bits? infinite?
- What happens when we run out of memory? How do we model this in the dynamic semantics?

# What is an address?

- How do we represent addresses, i.e., the result of the **alloc** operation?
- 64-bits? infinite?
- What happens when we run out of memory? How do we model this in the dynamic semantics?
- Assume infinite address space, i.e., an address is in  $\mathbb{N}$ .
- Out of heap memory will generate an exception: “exception(mem)”

# Using $H$

- $\text{alloc}(\tau)$  returns an address of proper size (or raises an exception)
- $H$  must keep track of next free address.

$$H: (\mathbb{N} \cup \{\text{next}\}) \rightarrow \text{Val}$$

- Extend all old rules with  $H$ ; which they leave unchanged, e.g.,

$$H; S; \eta \vdash e_1 \oplus e_2 \triangleright K \longrightarrow H; S; \eta \vdash e_1 \triangleright (\blacksquare \oplus e_2, K)$$

# Pointers

- null evaluates to 0

$$H; S; \eta \vdash \text{null} \triangleright K \longrightarrow H; S; \eta \vdash 0 \triangleright K$$

- alloc( $\tau$ ):

- returns a fresh address  $a$ ,
- updates the next address in the heap
- initializes the location to default for  $\tau$

$$H; S; \eta \vdash \text{alloc}(\tau) \triangleright K \longrightarrow$$

$$H[a \mapsto \text{default}(\tau), \text{next} \mapsto a + |\tau|]; S; \eta \vdash a \triangleright K$$

$a = H(\text{next})$

$H; S; \eta \vdash \text{alloc}(\tau) \triangleright K \longrightarrow$

$H[a \mapsto \text{default}(\tau), \text{next} \mapsto a + |\tau|]; S; \eta \vdash a \triangleright K$   
 $a = H(\text{next})$

- default( $\tau$ ): 0 for int, false for bool, null for ptr

- $|\tau|$  for x86-64:

- $|\text{int}| = 4$

- $|\text{bool}| = 4$

- $|\tau^*| = 8$

}



# Accessing Memory

- Dereferencing a pointer:

$$H; S; \eta \vdash_{\text{w}}^* e \triangleright K \longrightarrow H; S; \eta \vdash_{\text{w}} e \triangleright (* \downarrow \blacksquare, K)$$

# Accessing Memory

$\text{wp} = 1$

- Dereferencing a pointer:

$$H; S; \eta \vdash^* e \triangleright K \longrightarrow H; S; \eta \vdash e \triangleright (* \blacksquare, K)$$

- The interesting part:

$$H; S; \eta \vdash a \triangleright K \longrightarrow H; S; \eta \vdash H(a) \triangleright K \quad a \neq 0$$

$$H; S; \eta \vdash a \triangleright K \longrightarrow \text{exception(mem)} \quad a = 0$$

# Writing to the heap

- l-values and r-values
- l-values or destinations:

$$d ::= \underline{x} \mid * \underline{d}$$

- Typing is the same for all destinations:

$$\frac{\Gamma \vdash d : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{assign}(d, e) : [\tau']}$$

recall,  $[\tau']$ , is the return type of the function.

# Writing to the heap

- Distinguish between variables,  $x$ , which live on the stack,

$$\begin{array}{ll} H ; S ; \eta \vdash \text{assign}(\underline{x}, \underline{e}) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{assign}(x, \_), K) \\ H ; S ; \eta \vdash \underline{c} \triangleright (\text{assign}(\underline{x}, \underline{\_}), K) & \longrightarrow H ; S ; \eta[\underline{x \mapsto c}] \triangleright \text{nop} \blacktriangleright K \end{array}$$

# Writing to the heap

- Distinguish between variables,  $x$ , which live on the stack,

$$\begin{array}{ll} H ; S ; \eta \vdash \text{assign}(x, e) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{assign}(x, \_), K) \\ H ; S ; \eta \vdash c \triangleright (\text{assign}(x, \_), K) & \longrightarrow H ; S ; \eta[x \mapsto c] \triangleright \text{nop} \blacktriangleright K \end{array}$$

- and other destinations which live in the heap.

$$\begin{array}{ll} H ; S ; \eta \vdash \text{assign}(*d, e) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash d \triangleright (\text{assign}(*\_, e), K) \\ H ; S ; \eta \vdash a \triangleright (\text{assign}(*\_, e), K) & \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{assign}(*a, \_), K) \\ H ; S ; \eta \vdash c \triangleright (\text{assign}(*a, \_), K) & \longrightarrow \boxed{H[a \mapsto c]} S ; \eta \vdash \text{nop} \blacktriangleright K \quad (a \neq 0) \\ H ; S ; \eta \vdash \bar{c} \triangleright (\text{assign}(*a, \_), K) & \longrightarrow \text{exception(mem)} \quad (a = 0) \end{array}$$

# Writing to the heap

- left to right evaluation of address and r-value

$$\begin{array}{ll}
 H ; S ; \eta \vdash \text{assign}(*\underline{d}, e) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash d \triangleright (\text{assign}(*\_, e), K) \\
 H ; S ; \eta \vdash \underline{a} \triangleright (\text{assign}(*\underline{\_}, e), K) & \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{assign}(*a, \_), K)
 \end{array}$$

- Then making assignment (if  $a \neq 0$ )

# Writing to the heap

- left to right evaluation of address and rval

$$H ; S ; \eta \vdash \text{assign}(*d, e) \blacktriangleright K \quad \longrightarrow \quad H ; S ; \eta \vdash d \triangleright (\text{assign}(*_, e) , K)$$

$$H ; S ; \eta \vdash a \triangleright (\text{assign}(*_, e) , K) \quad \longrightarrow \quad H ; S ; \eta \vdash e \triangleright (\text{assign}(*a, _) , K)$$

- Then making assignment (if  $a \neq 0$ )

$$\begin{array}{ll} H ; S ; \eta \vdash c \triangleright (\text{assign}(*a, _) , K) & \longrightarrow \quad H[a \mapsto c] ; S ; \eta \vdash \text{nop} \blacktriangleright K \quad (a \neq 0) \\ H ; S ; \eta \vdash c \triangleright (\text{assign}(*a, _) , K) & \longrightarrow \quad \text{exception}(\text{mem}) \quad (a = 0) \end{array}$$

# Proper evaluation order

- `int* p = NULL;`  
`*p = 1/0;`

Handwritten notes for the first code snippet:

- $\vdash \text{assign}(*p, 1/0)$
- $\vdash p ; \text{assign}(*p, 1/0)$
- $\vdash \emptyset ; \text{assign}(*p, 1/0)$
- ~~$\vdash 1/0 ; \text{assign}(*p, 1/0)$~~
- $\text{exp (div by 0)}$

- `int**p = NULL;`  
`**p = 1/0;`

Handwritten notes for the second code snippet:

- $\vdash \text{assign}(**p, 1/0)$
- $\vdash \underline{*p} ; \text{assign}(*p, 1/0)$
- $\vdash \underline{p} ; \underline{*p} ; \text{assign}(*p, 1/0)$
- $\emptyset$



# Today

- Pointers
- The Heap and pointers
- Arrays
- Length & bounds checking
- Elaboration of +=, etc.

# Arrays: static semantics

- Extend types, expressions, and destinations

$$\begin{array}{lcl} \tau & ::= & \dots \mid \tau[] \\ e & ::= & \dots \mid \text{alloc\_array}(\tau, e) \mid e_1[e_2] \\ d & ::= & \dots \mid d[e] \end{array}$$

- Need typing rules for `alloc_array` and `e1[e2]`

$$\frac{\Gamma \vdash e : \text{int}}{\Gamma \vdash \text{alloc\_array}(\tau, e) : \tau[]}$$

$$\frac{\Gamma \vdash e_1 : \tau[] \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

# Allocating the array

$$\begin{array}{c} H ; S ; \eta \vdash \text{alloc\_array}(\tau, e) \triangleright K \\ \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc\_array}(\tau, \_), K) \end{array}$$

# Allocating the array

$$\begin{array}{l} H ; S ; \eta \vdash \text{alloc\_array}(\tau, e) \triangleright K \\ \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc\_array}(\tau, \_), K) \end{array}$$

$$\begin{array}{l} H ; S ; \eta \vdash n \triangleright (\text{alloc\_array}(\tau, \_), K) \\ \xrightarrow{\quad} H' ; S ; \eta \vdash a \triangleright K \end{array} \quad \underbrace{(n \geq 0)}$$

$$\longrightarrow \text{exception(mem)} \quad \underbrace{(n < 0)}$$

# Allocating the array



$$H ; S ; \eta \vdash \text{alloc\_array}(\tau, e) \triangleright K$$

$$\longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc\_array}(\tau, \_), K)$$

$$H ; S ; \eta \vdash n \triangleright (\text{alloc\_array}(\tau, \_), K)$$

$$\longrightarrow \left[ \begin{array}{l} H' ; S ; \eta \vdash a \triangleright K \\ a = H(\text{next}) \\ H' = H[\underline{a + 0|\tau|} \mapsto \text{default}(\tau), \dots, \\ \underline{a + (n - 1)|\tau|} \mapsto \text{default}(\tau), \text{next} \mapsto a + n|\tau|] \end{array} \right. \quad (n \geq 0)$$

$$\longrightarrow \text{exception}(\text{mem}) \quad (n < 0)$$

# Accessing the Array

- left to right evaluation of base address of array and index

$$\begin{array}{ll}
 H ; S ; \eta \vdash \underline{e_1[e_2]} \triangleright K & \longrightarrow H ; S ; \eta \vdash \underline{e_1} \triangleright (\underline{_[e_2]}, K) \\
 H ; S ; \eta \vdash \underline{a} \triangleright (\underline{_[e_2]}, K) & \longrightarrow H ; S ; \eta \vdash \underline{e_2} \triangleright (\underline{a[_]}, K)
 \end{array}$$

- Then, if in bounds, get the value

$$\begin{array}{ll}
 H ; S ; \eta \vdash \underline{i} \triangleright (\underline{a[_]}, K) & \longrightarrow H ; S ; \eta \vdash \underline{H(a + i|\tau|)} \triangleright K \\
 & \quad \underline{a \neq 0, 0 \leq i < \text{length}(a), a : \tau[]}
 \end{array}$$

- Or, generate an exception

$$\longrightarrow \text{exception(mem)}$$

$$a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a)$$

# Accessing the Array

- left to right evaluation of base address of array and index

$$\begin{array}{ll} H ; S ; \eta \vdash e_1[e_2] \triangleright K & \longrightarrow H ; S ; \eta \vdash e_1 \triangleright (\_ [e_2], K) \\ H ; S ; \eta \vdash \underline{a} \triangleright (\_ [e_2], K) & \longrightarrow H ; S ; \eta \vdash \underline{e_2} \triangleright (\underline{a[\_]}, K) \end{array}$$

- Then, if in bounds, get the value

$$\begin{array}{ll} H ; S ; \eta \vdash \underline{i} \triangleright (\underline{a[\_]}, K) & \longrightarrow H ; S ; \eta \vdash \underline{H(a + i[\tau])} \triangleright K \\ & a \neq 0, 0 \leq i < \text{length}(a), a : \tau[\_] \end{array}$$

- Or, generate an exception

$$\begin{array}{l} \longrightarrow \text{exception(mem)} \\ a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a) \end{array}$$

# Accessing the Array

- left to right evaluation of base address of array and index

$$\begin{array}{ll} H ; S ; \eta \vdash e_1[e_2] \triangleright K & \longrightarrow H ; S ; \eta \vdash e_1 \triangleright (\_ [e_2] , K) \\ H ; S ; \eta \vdash a \triangleright (\_ [e_2] , K) & \longrightarrow H ; S ; \eta \vdash e_2 \triangleright (a[\_] , K) \end{array}$$

- Then, if in bounds, get the value

$$H ; S ; \eta \vdash i \triangleright (a[\_] , K) \longrightarrow H ; S ; \eta \vdash H(\underline{a + i|\tau|}) \triangleright K$$

$a + (int \vdash i) \tau$   
 ~~$a + i|\tau|$~~

$$a \neq 0, 0 \leq i < \text{length}(a), a : \tau[ ]$$

- Or, generate an exception

$$\longrightarrow \text{exception}(\text{mem})$$

recall: alloc\_array( $\tau, e$ )

$$a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a)$$



# Bounds checking

- Constraints in design of  $length(a)$

no exception  
 # of elmts associated w/ array allocated to  
 'a'

stored in heap?



$$\begin{cases} a + 8 + \bar{u} + \text{...} \\ a + \bar{u} + 4 \end{cases}$$

# Bounds checking

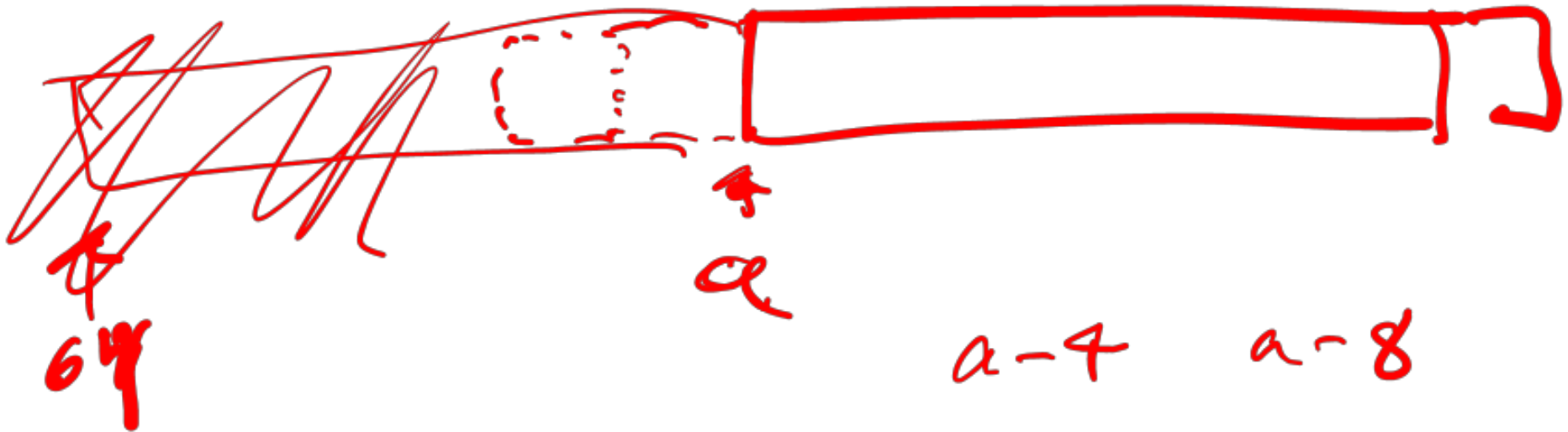
- Constraints in design of  $length(a)$

→ ○ Be able to find length of array given  $a$

→ ○ Minimize code size

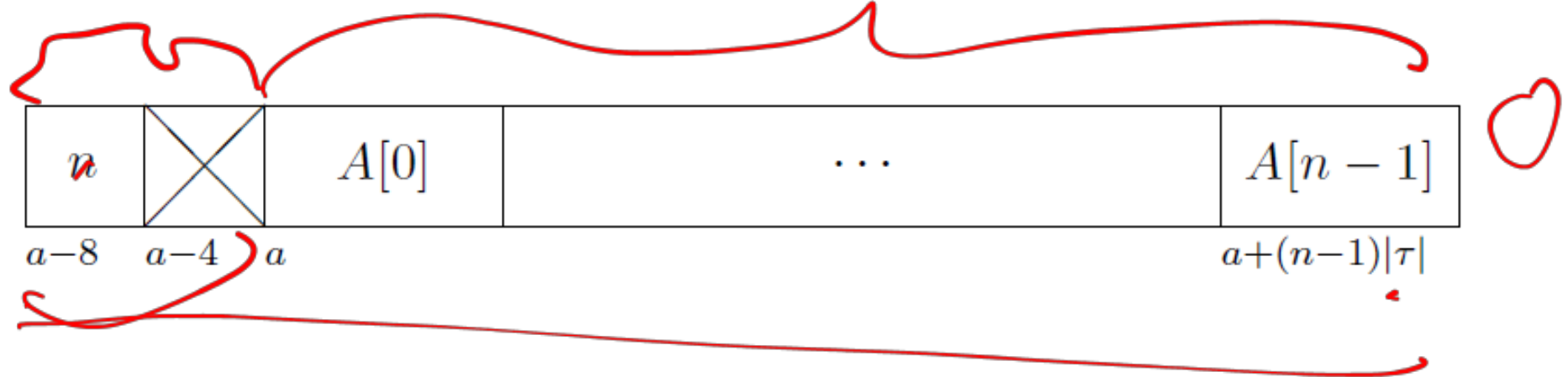
○ Alignment (padding, etc.)

→ ○ Inter-operability



# Bounds checking

- Must store length in heap.



- Rationale for storing length at  $a-8$ ?

# Writing to the array

$$\begin{array}{ll}
 H ; S ; \eta \vdash \text{assign}(d[e_2], e_3) \blacktriangleright K & \longrightarrow H ; S ; \eta \vdash d \triangleright (\text{assign}(\_ [e_2], e_3) , K) \\
 \\
 H ; S ; \eta \vdash a \triangleright (\text{assign}(\_ [e_2], e_3) , K) & \longrightarrow H ; S ; \eta \vdash e_2 \triangleright (\text{assign}(a[\_], e_3) , K) \\
 \\
 H ; S ; \eta \vdash i \triangleright (\text{assign}(a[\_], e_3) , K) & \longrightarrow H ; S ; \eta \vdash e_3 \triangleright (\text{assign}(\underline{a + i|\tau|}, \_) , K) \\
 & \qquad a \neq 0, 0 \leq i < \text{length}(a), a : \tau[] \\
 & \longrightarrow \text{exception(mem)} \\
 & \qquad a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a) \\
 \\
 H ; S ; \eta \vdash c \triangleright (\text{assign}(\underline{b}, \_) , K) & \longrightarrow H[\underline{b \mapsto c}] ; S ; \eta \vdash \text{nop} \blacktriangleright K
 \end{array}$$

# one caveat

$$H ; S ; \eta \vdash \text{assign}(d[e_2], e_3) \blacktriangleright K \quad \longrightarrow \quad H ; S ; \eta \vdash d \triangleright (\text{assign}(\_ [e_2], e_3) , K)$$

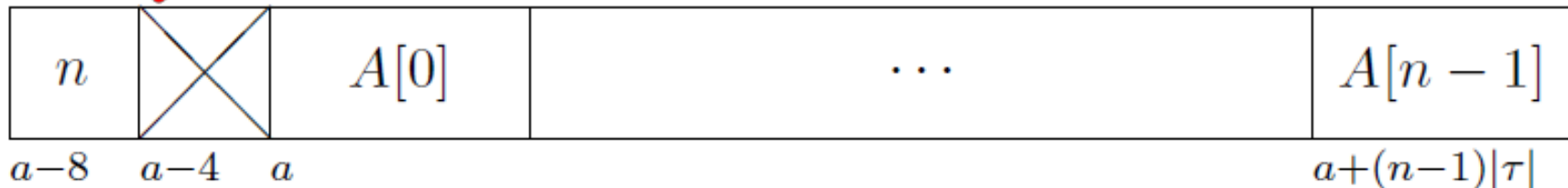
$$H ; S ; \eta \vdash a \triangleright (\text{assign}(\_ [e_2], e_3) , K) \quad \longrightarrow \quad H ; S ; \eta \vdash e_2 \triangleright (\text{assign}(a[\_], e_3) , K)$$

$$H ; S ; \eta \vdash i \triangleright (\text{assign}(a[\_], e_3) , K) \quad \longrightarrow \quad H ; S ; \eta \vdash e_3 \triangleright (\text{assign}(a + i|\tau|, \_) , K)$$

$a \neq 0, 0 \leq i < \text{length}(a) \quad a : \tau[]$   
 $\text{exception(mem)}$   
 $a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a)$

$$H ; S ; \eta \vdash c \triangleright (\text{assign}(b, \_) , K) \quad \longrightarrow \quad H[b \mapsto c] ; S ; \eta \vdash \text{nop} \blacktriangleright K$$

~~$(-8)$~~   ~~$(\text{offset} \dots)$~~



# Code Generation

*codegen( $\tau$ )*

- For access:  $e_1[e_2]$  where  $e_1:\tau[]$  and  $|\tau|=k$

*H/s/j/t  $e, [e_2] \triangleright k$*

*$a_1 \leftarrow \text{codegen}(e_1)$   
 $\text{codegen}(e_2)$*

*fresh a  
fresh i*

*$\text{cmp}(a, \emptyset)$*

*JE error*

*$a_1 = a - 8$*

*$\text{cmp}(+a_1, i)$*

*~~JE error~~*

*JG error*

*$b = a + i * |\tau|$*

# Code Generation

- For access:  $e_1[e_2]$  where  $e_1:\tau[]$  and  $|\tau|=k$

```
cogen( $e_1, a$ )           ( $a$  new)
cogen( $e_2, i$ )           ( $i$  new)
 $a_1 \leftarrow a - 8$ 
 $t_2 \leftarrow M[a_1]$ 
if ( $i < 0$ ) goto error
if ( $i \geq t_2$ ) goto error
 $a_3 \leftarrow i * \$k$ 
 $a_4 \leftarrow a + a_3$ 
 $t_5 \leftarrow M[a_4]$ 
```

not quite

# Elaboration

- $x = x + e$  is no longer always valid for  $x += e$

exp += e      a[5] += 2  
pure int      a[f(x)] += 2



# Elaboration

- $x = x + e$  is no longer always valid for  $x += e$
- next time introduce structure and &