Mutable Store

15-411/15-611 Compiler Design

Seth Copen Goldstein

March 11, 2026

Today

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

© 2019-21 Goldstein

Adding a pointer

- Extend types

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

- Extend expressions
  - **alloc(τ)**: allocate a heap cell to hold a value of τ
  - ***e**: dereference a pointer to get value at *e*
  - **null**: special null pointer

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

© 2019-21 Goldstein

Typing rules

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

- A freshly allocated cell has type "pointer to τ"

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

- if *e* has type "pointer to τ," then $*e$ has type "τ"

not quite

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

- What type should null have?

© 2019-21 Goldstein

The type of null?

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

© 2019-21 Goldstein

Equality for pointers?

- Can we compare expression of τ* and σ*:
  - if $\tau = \sigma$:
  - if $\tau \neq \sigma$:
  - What about `int* p; … if (p==null) …`
- null is given type of "any*"
- And, implicitly converted to τ* as needed

© 2019-21 Goldstein

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

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

© 2019-21 Goldstein

Polymorphic equality

$$\frac{\Gamma \vdash e_1 : \tau* \quad \Gamma \vdash e_2 : \tau*}{\Gamma \vdash e_1 == e_2 : \text{bool}}$$

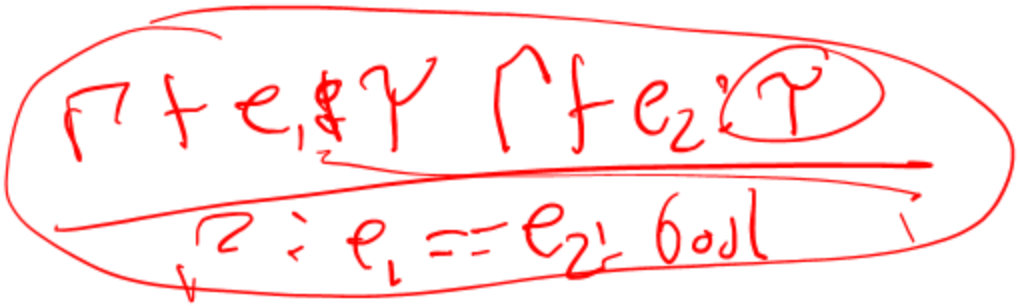
- We can compare two pointers

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 == e_2 : \text{bool}}$$

- We can compare two ints

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 == e_2 : \text{bool}}$$

- We can compare two booleans



© 2019-21 Goldstein

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

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

- Have to make sure introducing any* is safe

© 2019-21 Goldstein

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

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

- Have to make sure introducing any* is safe

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

© 2019-21 Goldstein

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

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

- Can't allow **`*null`**

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

© 2019-21 Goldstein

Typing rules (revised)

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

- A freshly allocated cell has type "pointer to τ"

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

- if *e* has type "pointer to τ," and *e* isn't null, then **e* has type "τ"

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

- null has the indefinite type

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

- Implicit coercion

© 2019-21 Goldstein

Representing the Heap

Evaluation of expression e in the context of

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

$$H; S; \eta \vdash e \rhd K$$

- alloc(τ) returns an unused address in *H* (the heap) which can store a value of τ

© 2019-21 Goldstein

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?

© 2019-21 Goldstein

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)”

© 2019-21 Goldstein

Using *H*

- alloc(τ) 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.,

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

© 2019-21 Goldstein

Pointers

- null evaluates to 0

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

- alloc(τ):
  - returns a fresh address *a*,
  - updates the next address in the heap
  - initializes the location to default for τ

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

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

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

© 2019-21 Goldstein

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

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

- $\text{default}(\tau)$: 0 for int, false for bool, null for ptr
- $|\tau|$ for x86-64:
  - |int| = 4
  - |bool| = 4
  - $|\tau^*|$ = 8

© 2019-21 Goldstein

Accessing Memory

- Dereferencing a pointer:

$$H; S; \eta \vdash * \mathrm{e} \rhd K \longrightarrow H; S; \eta \vdash e \rhd (* \blacksquare, K)$$

$H; S; \eta \vdash a \rhd (* \blacksquare, K) \rightarrow H; S; \eta \vdash H(a) \rhd K$

© 2019-21 Goldstein

Accessing Memory

- Dereferencing a pointer:

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

- The interesting part:

$$H; S; \eta \vdash a \triangleright (* \blacksquare, K) \longrightarrow H; S; \eta \vdash H(a) \triangleright K \qquad a \neq 0$$

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

© 2019-21 Goldstein

Writing to the heap

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

$$d ::= x \mid *d$$

- Typing is the same for all destinations:

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

recall, [τ'], is the return type of the function.

© 2019-21 Goldstein

Writing to the heap

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

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{assign}(x, \_)\ ,\ K)$$

$$H\ ;\ S\ ;\ \eta \vdash c \rhd (\mathsf{assign}(x, \_)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta[x \mapsto c] \rhd \mathsf{nop} \blacktriangleright K$$

© 2019-21 Goldstein

Writing to the heap

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

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \triangleright (\mathsf{assign}(x, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta[x \mapsto c] \triangleright \mathsf{nop} \blacktriangleright K$$

- and other destinations which live in the heap.

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

© 2019-21 Goldstein

Writing to the heap

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

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \triangleright (\mathsf{assign}(x, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta[x \mapsto c] \triangleright \mathsf{nop} \blacktriangleright K$$

- and other destinations which live in the heap.

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

© 2019-21 Goldstein

Writing to the heap

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

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{assign}(x, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \rhd (\mathsf{assign}(x, \_)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta[x \mapsto c] \rhd \mathsf{nop} \blacktriangleright K$$

- and other destinations which live in the heap.

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(*d, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash d \rhd (\mathsf{assign}(*\_, e)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash a \rhd (\mathsf{assign}(*\_, e)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{assign}(*a, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \rhd (\mathsf{assign}(*a, \_)\ ,\ K) \longrightarrow H[a \mapsto c]\ ;\ S\ ;\ \eta \vdash \mathsf{nop} \blacktriangleright K \quad (a \neq 0)$$

Writing to the heap

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

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \triangleright (\mathsf{assign}(x, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta[x \mapsto c] \triangleright \mathsf{nop} \blacktriangleright K$$

- and other destinations which live in the heap.

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(*d, e) \blacktriangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash d \triangleright (\mathsf{assign}(*\_, e)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash a \triangleright (\mathsf{assign}(*\_, e)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e \triangleright (\mathsf{assign}(*a, \_)\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(*a, \_)\ ,\ K) \longrightarrow H[a \mapsto c]\ ;\ S\ ;\ \eta \vdash \mathsf{nop} \blacktriangleright K \quad (a \neq 0)$$
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(*a, \_)\ ,\ K) \longrightarrow \mathsf{exception}(\mathsf{mem}) \quad (a = 0)$$

© 2019-21 Goldstein

Writing to the heap

- left to right evaluation of address and rval

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

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

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

$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(*a, \_)\ ,\ K) \longrightarrow H[a \mapsto c]\ ;\ S\ ;\ \eta \vdash \mathsf{nop} \blacktriangleright K \quad (a \neq 0)$$

$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(*a, \_)\ ,\ K) \longrightarrow \mathsf{exception}(\mathsf{mem}) \quad (a = 0)$$

© 2019-21 Goldstein

Proper evaluation order

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

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

© 2019-21 Goldstein

Today

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

© 2019-21 Goldstein

Arrays: static semantics

- Extend types, expressions, and destinations

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \tau[\,] \\ e & ::= & \ldots \mid \mathsf{alloc\_array}(\tau, e) \mid e_1[e_2] \\ d & ::= & \ldots \mid d[e] \end{array}$$

- Need typing rules for alloc_array and $e_1[e_2]$

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

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

© 2019-21 Goldstein

Allocating the array

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{alloc\_array}(\tau, e) \rhd K$$
$$\longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K)$$

© 2019-21 Goldstein

Allocating the array

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

$$\longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K)$$

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

$$\longrightarrow \qquad H'\ ;\ S\ ;\ \eta \vdash a \rhd K \qquad (n \geq 0)$$

$$\longrightarrow \qquad \mathsf{exception}(\mathsf{mem}) \qquad (n < 0)$$

© 2019-21 Goldstein

Allocating the array

(η, K)

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

$$\longrightarrow \quad H\ ;\ S\ ;\ \eta \vdash e \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K)$$

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

$$\longrightarrow \quad H'\ ;\ S\ ;\ \eta \vdash a \rhd K \qquad (n \geq 0)$$

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

$$H' = H[a + 0|\tau| \mapsto \mathsf{default}(\tau), \ldots,$$

$$a + (n-1)|\tau| \mapsto \mathsf{default}(\tau), \mathsf{next} \mapsto a + n|\tau|]$$

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

$a[0]$ $\quad a+8+0*|\tau|$ $\quad a+n|\tau|+8$

© 2019-21 Goldstein

Accessing the Array

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

$$H\,;S\,;\eta \vdash e_1[e_2] \rhd K \longrightarrow H\,;S\,;\eta \vdash e_1 \rhd (\_[e_2]\,,K)$$
$$H\,;S\,;\eta \vdash a \rhd (\_[e_2]\,,K) \longrightarrow H\,;S\,;\eta \vdash e_2 \rhd (a[\_]\,,K)$$

© 2019-21 Goldstein

Accessing the Array

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

$$H\,;S\,;\eta \vdash e_1[e_2] \rhd K \longrightarrow H\,;S\,;\eta \vdash e_1 \rhd (\_[e_2]\,,K)$$
$$H\,;S\,;\eta \vdash a \rhd (\_[e_2]\,,K) \longrightarrow H\,;S\,;\eta \vdash e_2 \rhd (a[\_]\,,K)$$

- Then, if in bounds, get the value

$$H\,;S\,;\eta \vdash i \rhd (a[\_]\,,K) \longrightarrow H\,;S\,;\eta \vdash H(a+i|\tau|) \rhd K$$
$$a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,]$$

- Or, generate an exception

$$\longrightarrow \mathsf{exception(mem)}$$
$$a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)$$

© 2019-21 Goldstein

Accessing the Array

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

$$H\ ;\ S\ ;\ \eta \vdash e_1[e_2] \rhd K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_1 \rhd (\_[e_2]\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash a \rhd (\_[e_2]\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_2 \rhd (a[\_]\ ,\ K)$$

- Then, if in bounds, get the value

$$H\ ;\ S\ ;\ \eta \vdash i \rhd (a[\_]\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash H(a + i[\tau]) \rhd K$$

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

- Or, generate an exception

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

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

© 2019-21 Goldstein

Accessing the Array

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

$$H\ ;\ S\ ;\ \eta \vdash e_1[e_2] \triangleright K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_1 \triangleright (\_[e_2]\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash a \triangleright (\_[e_2]\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_2 \triangleright (a[\_]\ ,\ K)$$

- Then, if in bounds, get the value

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

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

- Or, generate an exception

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

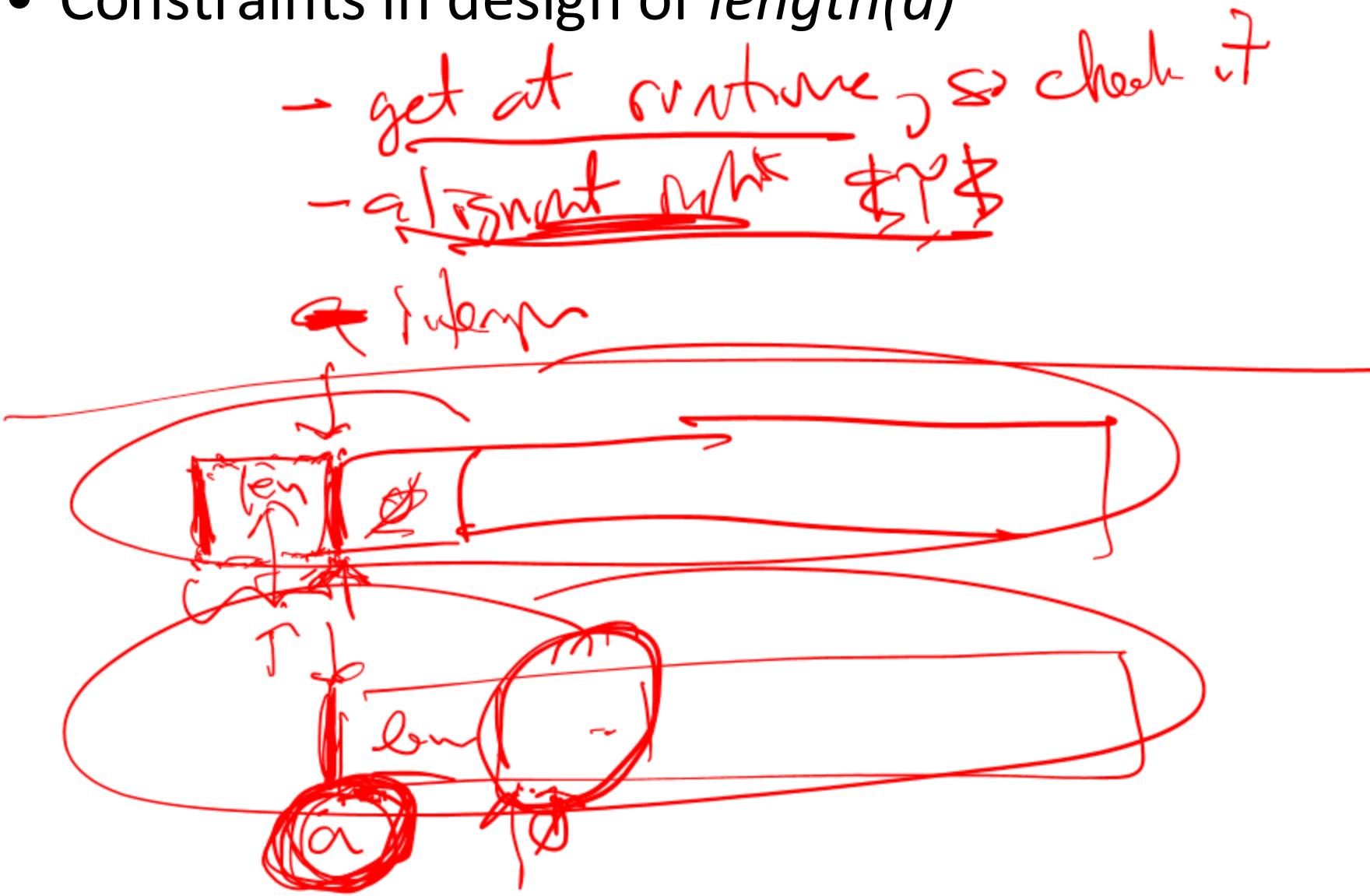
$$a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)$$

recall: alloc_array(τ,e)

© 2019-21 Goldstein

Bounds checking

- Constraints in design of *length(a)*



© 2019-21 Goldstein

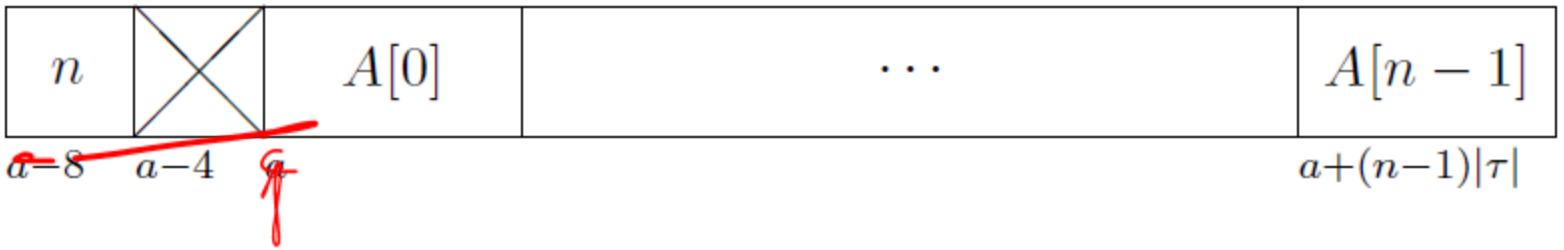
Bounds checking

- Constraints in design of *length(a)*
  - Be able to find length of array given *a*
  - Minimize code size
  - Alignment (padding, etc.)
  - Inter-operability

© 2019-21 Goldstein

Bounds checking

- Must store length in heap.



- Rationale for storing length at a-8?

© 2019-21 Goldstein

Writing to the array

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

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

$$H\ ;\ S\ ;\ \eta \vdash i \triangleright (\mathsf{assign}(a[\_], e_3)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_3 \triangleright (\mathsf{assign}(a + i|\tau|, \_)\ ,\ K)$$
$$a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,]$$

$$\longrightarrow \mathsf{exception}(\mathsf{mem})$$
$$a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)$$

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

© 2019-21 Goldstein

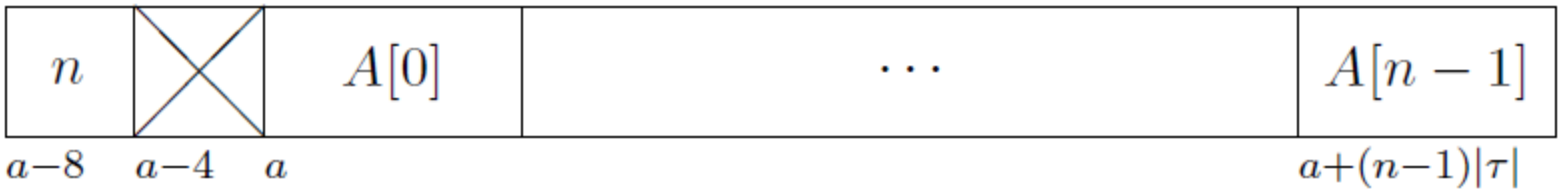
one caveat

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

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

$$H\ ;\ S\ ;\ \eta \vdash i \triangleright (\mathsf{assign}(a[\_], e_3)\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_3 \triangleright (\mathsf{assign}(a + i|\tau|, \_)\ ,\ K)$$
$$a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,]$$
$$\longrightarrow \mathsf{exception}(\mathsf{mem})$$
$$a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)$$

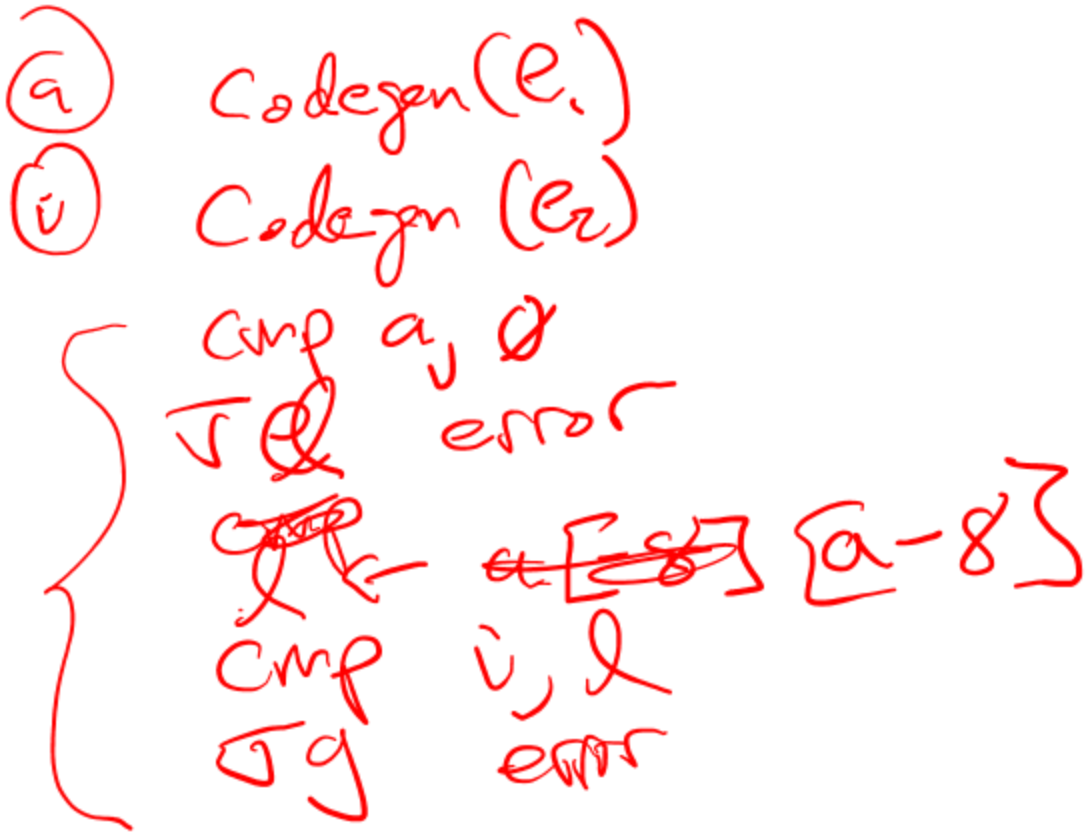
$$H\ ;\ S\ ;\ \eta \vdash c \triangleright (\mathsf{assign}(b, \_)\ ,\ K) \longrightarrow H[b \mapsto c]\ ;\ S\ ;\ \eta \vdash \mathsf{nop} \blacktriangleright K$$



© 2019-21 Goldstein

Code Generation

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



© 2019-21 Goldstein

Accessing the Array

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

$$H\ ;\ S\ ;\ \eta \vdash e_1[e_2] \rhd K \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_1 \rhd (\_[e_2]\ ,\ K)$$
$$H\ ;\ S\ ;\ \eta \vdash a \rhd (\_[e_2]\ ,\ K) \longrightarrow H\ ;\ S\ ;\ \eta \vdash e_2 \rhd (a[\_]\ ,\ K)$$

- Then, if in bounds, get the value

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

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

- Or, generate an exception

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

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

© 2019-21 Goldstein

Code Generation

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

$\text{cogen}(e_1, a)$ (a new)
$\text{cogen}(e_2, i)$ (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
$a_3 \leftarrow i * \$k$
$a_4 \leftarrow a + a_3$
$t_5 \leftarrow M[a_4]$

© 2019-21 Goldstein

Elaboration

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

© 2019-21 Goldstein

Elaboration

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

© 2019-21 Goldstein

Coding w/AI

- Production
  - Are you being efficient?
  - Are you understanding the code?
  - Will it be maintainable?
- Education
  - Are you learning?

© 2019-21 Goldstein