

Mutable Store

15-411/15-611 Compiler Design

Seth Copen Goldstein

October 19, 2021

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

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

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 τ ”
- if e has type “pointer to τ ,” then $*e$ has type “ τ ”
- What type should null have?

The type of null?

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

Equality for pointers?

- Can we compare 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

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

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

- 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 τ ”
- if e has type “pointer to τ ,” and e isn’t null, then $*e$ has type “ τ ”
- null has the indefinite type
- Implicit coercion

Representing the Heap

Evaluation of expression e in the context of

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

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

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

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.

$$H: (\text{NU}\{\text{next}\}) \rightarrow \text{Val}$$

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

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

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

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

- $\text{default}(\tau)$: 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:

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

Accessing Memory

- 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:

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

- 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,

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

Writing to the heap

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

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

- and other destinations which live in the heap.

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

Writing to the heap

- left to right evaluation of address and r-value

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

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

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

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 $a \neq 0$)

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

Proper evaluation order

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

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

Today

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

Arrays: static semantics

- Extend types, expressions, and destinations

$$\tau ::= \dots \mid \tau[]$$
$$e ::= \dots \mid \text{alloc_array}(\tau, e) \mid e_1[e_2]$$
$$d ::= \dots \mid d[e]$$

- 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

$$\begin{aligned} H ; S ; \eta \vdash \text{alloc_array}(\tau, e) \triangleright K \\ \longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc_array}(\tau, _), K) \end{aligned}$$

Allocating the array

$$H ; S ; \eta \vdash \text{alloc_array}(\tau, e) \triangleright K$$
$$\longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc_array}(\tau, _), K)$$
$$H ; S ; \eta \vdash n \triangleright (\text{alloc_array}(\tau, _), K)$$
$$\longrightarrow H' ; S ; \eta \vdash a \triangleright K \quad (n \geq 0)$$
$$\longrightarrow \text{exception}(\text{mem}) \quad (n < 0)$$

Allocating the array

$$H ; S ; \eta \vdash \text{alloc_array}(\tau, e) \triangleright K$$
$$\longrightarrow H ; S ; \eta \vdash e \triangleright (\text{alloc_array}(\tau, _), K)$$
$$H ; S ; \eta \vdash n \triangleright (\text{alloc_array}(\tau, _), K)$$
$$\longrightarrow H' ; S ; \eta \vdash a \triangleright K \quad (n \geq 0)$$
$$a = H(\text{next})$$
$$H' = H[a + 0|\tau| \mapsto \text{default}(\tau), \dots,$$
$$a + (n - 1)|\tau| \mapsto \text{default}(\tau), \text{next} \mapsto a + n|\tau|]$$
$$\longrightarrow \text{exception}(\text{mem}) \quad (n < 0)$$

Accessing the Array

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

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

- Then, if in bounds, get the value

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

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

- Or, generate an exception

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

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

Accessing the Array

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

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

- Then, if in bounds, get the value

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

- 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

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

- Then, if in bounds, get the value

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

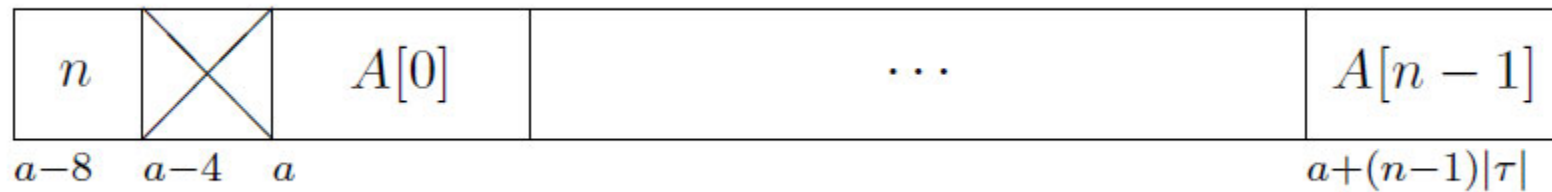
- Or, generate an exception

recall: alloc_array(τ, e)

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

Bounds checking

- Must store length in heap.



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

Writing to the array

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

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

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

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

one caveat

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

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

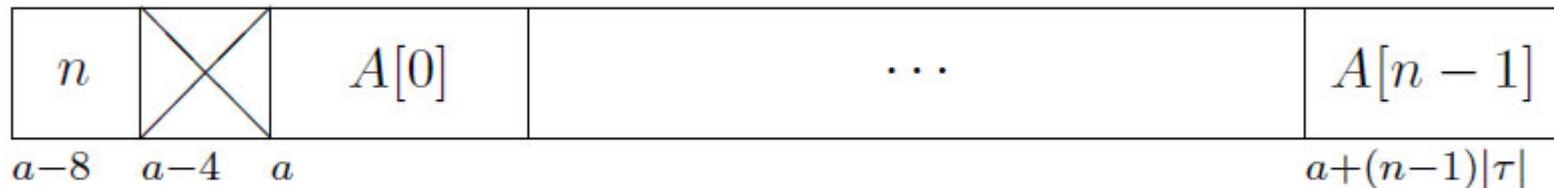
$$H ; S ; \eta \vdash i \triangleright (\text{assign}(a[_], e_3) , K) \quad \longrightarrow \quad H ; S ; \eta \vdash e_3 \triangleright (\text{assign}(a + i|\tau|, _) , K)$$

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

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

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

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



Code Generation

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

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 $x += e$

Elaboration

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