## 15-411: Mutable Store

Jan Hoffmann

# Pointers and Arrays

We will see how static and dynamic semantics make it easy to introduce and specify advanced language features

- Static semantics of pointers
- Dynamic semantics of pointers
- Static semantics of arrays
- Dynamic semantics of arrays

# Recap

### Dynamic Semantics - Configurations Dupopio Composico Configurati **Buyilianing Statical Response Property**



*·* ; *·* ` main( ) B *·* (initial state)

value(c) – Final state, return a value  
exception(
$$
E
$$
) – Final state, report an error



## **Definitions**

- $S; \eta \vdash c_2 \rhd (c_1 \odot \ldots K)$   $\longrightarrow$   $S; \eta \vdash c \rhd K$   $(c = c_1 \odot c_2)$ <br>  $S; \eta \vdash c_2 \rhd (c_1 \odot \ldots K)$   $\longrightarrow$  exception(arith)  $(c_1 \odot c_2 \text{ un})$
- $S$ ;  $\eta \vdash e_1$  &&  $e_2 \rhd K$   $\longrightarrow$  $S$ ;  $\eta$   $\vdash$  false  $\triangleright$  ( $\bot$  &&  $e_2$ ,  $K$ )  $S$ ;  $\eta$   $\vdash$  true  $\triangleright$  ( $\bot$  &&  $e_2$  ,  $K$ )

- $S; \eta \vdash e_1 \odot e_2 \rhd K$ <br>  $S; \eta \vdash e_1 \rhd (\_ \odot e_2, K)$ <br>  $S; \eta \vdash e_2 \rhd (c_1 \odot \_ K)$  $S; \eta \vdash c_1 \triangleright (\_ \odot e_2, K) \longrightarrow S; \eta \vdash e_2 \triangleright (c_1 \odot \_ , K)$ <br>  $S; \eta \vdash c_2 \triangleright (c_1 \odot \_ , K) \longrightarrow S; \eta \vdash c \triangleright K \qquad (c = c_1$ 
	- $\longrightarrow$  exception(arith)  $(c_1 \odot c_2 \text{ undefined})$

$$
S; \eta \vdash e_1 \rhd (\_ \& e_2 \;, K)
$$

$$
\rightarrow \qquad S \mathrel{;} \eta \vdash \mathsf{false} \rhd K
$$

$$
\rightarrow \qquad S \mathrel{;} \eta \vdash e_2 \rhd K
$$

$$
S: \eta \vdash x \rhd K \qquad \longrightarrow \qquad S: \eta \vdash \eta(x) \rhd K
$$

## **Transitions: Expressions S** is true B (*S*  $\mathsf{S}$   $\mathsf{S}$  )  $\mathsf{S}$   $\math$

- 
- $S$ ;  $\eta \vdash \mathsf{nop} \blacktriangleright (s, K)$ <br> $S$ ;  $\eta \vdash \mathsf{assign}(x, e) \blacktriangleright K$
- 
- 

 $S$ ;  $\eta \vdash$  decl $(x, \tau, s) \blacktriangleright K$   $\longrightarrow$ 

- $S: \eta \vdash \mathsf{assert}(e) \blacktriangleright K$ <br>  $S: \eta \vdash \mathsf{true} \rhd (\mathsf{assert}(\ ) \cdot K)$   $\longrightarrow$  $S$ ;  $\eta \vdash \textsf{true} \rhd (\textsf{assert}(\_), K) \longrightarrow$ <br> $S : \eta \vdash \textsf{false} \rhd (\textsf{assert}(\_), K) \longrightarrow$  $S$ ;  $\eta$   $\vdash$  false  $\triangleright$  (assert(\_),  $K$ )  $\longrightarrow$  exception(abort)
- $S: \eta \vdash \text{if}(e, s_1, s_2) \blacktriangleright K$ <br>  $S: \eta \vdash t$ rue  $\triangleright$  (if(\_, s<sub>1</sub>, s<sub>2</sub>), K)<br>  $\longrightarrow$   $S: \eta \vdash s_1 \blacktriangleright K$  $S$ ;  $\eta$   $\vdash$  true  $\triangleright$  (if(\_, s<sub>1</sub>, s<sub>2</sub>), K)  $S$ ;  $\eta$   $\vdash$  s<sub>1</sub>  $\blacktriangleright$  K<br>  $S$ ;  $\eta$   $\vdash$  false  $\triangleright$  (if(\_, s<sub>1</sub>, s<sub>2</sub>), K)  $S$ ;  $\eta$   $\vdash$  s<sub>2</sub>  $\blacktriangleright$  K  $S$ ;  $\eta$   $\vdash$  false  $\triangleright$  (if(\_*, s*<sub>1</sub>*, s*<sub>2</sub>)*, K*)

 $S; \eta \vdash \mathsf{seq}(s_1, s_2) \blacktriangleright K$   $\longrightarrow$   $S; \eta \vdash s_1 \blacktriangleright (s_2, K)$ <br>  $S; \eta \vdash \mathsf{nop} \blacktriangleright (s, K)$   $\longrightarrow$   $S; \eta \vdash s \blacktriangleright K$  $S; \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K$   $S; \eta \vdash e \rhd (\mathsf{assign}(x, \_), K)$   $S; \eta \vdash c \rhd (\mathsf{assign}(x, \_), K)$  $\longrightarrow$   $S$ ;  $\eta[x \mapsto c]$   $\vdash$  nop  $\blacktriangleright$  *K* 

$$
S: \eta[x \mapsto \text{nothing}] \vdash s \blacktriangleright K
$$

$$
S: \eta \vdash e \rhd (\text{assert}(\_), K)
$$
  

$$
S: \eta \vdash \text{nop} \blacktriangleright K
$$
  
excentration(short)

 $S$ ;  $\eta \vdash \text{while}(e, s) \blacktriangleright K$   $\longrightarrow$   $S$ ;  $\eta \vdash \text{if}(e, \text{seq}(s, \text{while}(e, s)), \text{nop}) \blacktriangleright K$ 

## **Transitions: Statements**  $\frac{1}{2}$  **s**  $\frac{1}{2}$   $\frac{1}{2$

$$
S; \eta \vdash f(e_1, e_2) \rhd K \longrightarrow S; \eta \vdash e_1 \rhd (f(., e_2), K) \nS; \eta \vdash c_1 \rhd (f(., e_2), K) \longrightarrow S; \eta \vdash e_2 \rhd (f(c_1, .), K) \nS; \eta \vdash c_2 \rhd (f(c_1, .), K) \longrightarrow (S, \langle \eta, K \rangle) ; [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright.
$$

$$
S: \eta \vdash f() \rhd K \longrightarrow (S, \langle \eta, K \rangle) : \vdash s \blacktriangleright \cdot
$$
  
(*given that f is a*)

$$
S; \eta \vdash return(e) \blacktriangleright K \longrightarrow S; \eta \vdash e \triangleright (return(\_) , K) (S, \langle \eta', K' \rangle) ; \eta \vdash v \triangleright (return(\_) , K) \longrightarrow S; \eta' \vdash v \triangleright K' \cdot; \eta \vdash c \triangleright (return(\_) , K) \longrightarrow value(c)
$$

#### Special case: returning void  $\Omega$  ;  $\Omega$  is  $\Omega$  is  $\Omega$  and  $\Omega$  if  $\Omega$  is  $\Omega$  and  $\Omega$  is  $\Omega$ **SE:** TELUMING VOIG  $S$  posial case; returning void *S* pecial case: returning void  $S$  posial case; returning void Special case: returning void<br>
Special case: returning void

$$
S \ , \ \langle \eta', K' \rangle \ ; \ \eta \vdash \mathsf{nop} \ \blacktriangleright \ \cdot \quad \ \ \longrightarrow \quad \ \ S \ ; \ \eta' \vdash \mathsf{nothing} \ \rhd K'
$$

### Expressions as statements: return statement statement statement of the Expressions as statements:

$$
S: \eta \vdash e \blacktriangleright K \longrightarrow S: \eta \vdash e \triangleright (\text{discard}, K) S: \eta \vdash v \triangleright (\text{discard}, K) \longrightarrow S: \eta \vdash \text{nop} \blacktriangleright K
$$

#### Transitions: Functions  $\overline{F}$ *S* ; ⌘ ` *c*<sup>1</sup> B (*f*(\_*, e*2) *, K*) ! *S* ; ⌘ ` *e*<sup>2</sup> B (*f*(*c*1*,* \_) *, K*)  $T_{\text{FOP}}$   $\alpha$ <sup>1</sup>,  $\alpha$ <sup>1</sup>,  $\alpha$ <sub>2</sub>,  $\alpha$ <sup>1</sup>,  $\alpha$ <sub>2</sub>,  $\alpha$ <sub>1</sub> **S**  $\alpha$  *c <i>c*<sub>1</sub> *c*<sub>*c*</sub> *c*<sub>1</sub> *c*<sub>1</sub> *c*<sub>1</sub> *c*<sub>2</sub> B (*c*<sub>1</sub>*c*) *, C<sub>1</sub> B (<i>c*<sub>1</sub>*c*) *, C<sub>1</sub>* B (*c*<sub>1</sub>*c*) *, C<sub>1</sub>* B (*c*<sub>1</sub>*c*) *, C<sub>1</sub>* B (*c*<sub>1</sub>*c*) *, C<sub>1</sub>* B (*c*<sub>1</sub>*c*) *, C<sub>1</sub> , C<sub>1</sub> , C<sub>1</sub> , C<sub>1</sub>*  $\sum_{i=1}^{n}$ **S**  $\alpha$  **C**  $\beta$  *c*  $\beta$  *<i>c*  $\beta$  *c*  $\beta$  *<i>c*  $\beta$  *c*  $\beta$

 $S; \eta \vdash c_2 \rhd (f(c_1, \square), K)$   $\longrightarrow$   $(S, \langle \eta, K \rangle)$ ;  $[x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright$ (given that  $f$  is defined as  $f(x_1, x_2)$  $\{s\}$ )  $S \cdot n \vdash f() \triangleright K$   $\longrightarrow$   $(S \ (n \ K)) \cdot \vdash s \blacktriangleright$  $S: \eta \vdash c_2 \rhd (f(c_1, \ldots), K)$ <br>  $S: \eta \vdash c_2 \rhd (f(c_1, \ldots), K)$ <br>  $\longrightarrow$   $(S, \langle \eta, K \rangle) : [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright$  $\left(\varphi, \gamma, \varphi_1, \varphi_2, \varphi_3, \varphi_4\right)$ <br>(given that f is defined as  $f(x_1, x_2)$ {s})  $\chi$ iven ma j rompinent wo $\chi$  $\psi$ <sub>1</sub>, $\omega$ <sub>2</sub>) $\psi$ 

*(given that f is defined as f*( )*{s})* i ; ⌘ ` *v* B (return(\_) *, K*) ! *S* ; ⌘<sup>0</sup> ` *v* B *K*<sup>0</sup>  $S; \eta \vdash f() \triangleright K \longrightarrow (S, \langle \eta, K \rangle) ; \vdash s \blacktriangleright$  $g$ iven thut  $f$  is defined us  $f(f)$ 

$$
e) \triangleright K \longrightarrow S; \eta \vdash e \triangleright (\text{return}(\_), K) \longrightarrow S; \eta' \vdash v \triangleright K'
$$
  
\n
$$
\text{urn}(\_), K) \longrightarrow S; \eta' \vdash v \triangleright K'
$$
  
\n
$$
\text{win}(\_), K) \longrightarrow \text{value}(c)
$$

# Static semantics of pointers

# Static Semantics of Pointers

Extend types with pointer types: **EXTEND CYPES WITH POINTER CYPES. 2 Pointers** 

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

Extend expressions with allocations, dereference, and null pointers: pointer to obtain the stored value. exterio expressions with allocations, deference, and null pointers:

 $e ::= \ldots | \mathsf{alloc}(\tau) | *e | \mathsf{null}$ a value of type  $\alpha$  , we have a distinguished null pointer, and we can dereference a distinguished null point a value of type  $\alpha$  , we have a distinguished null pointer, and we can dereference and we can dereference and a value of type  $\alpha$  , we have a distinguished null pointer, and we can dereference and we can dereference and

 $\overline{\phantom{a}}$  *e* ion  $\overline{\phantom{a}}$  if  $\overline{\phantom{a}}$  and  $\overline{\phantom{a}}$  if  $\overline{\phant$  $\Gamma \vdash e : \tau *$  $\Gamma \vdash e : \tau$ We add the following typing rules for expressions: *e* ::= *... |* alloc(⌧ ) *|* ⇤*e |* null we dud the following typing rules *e* ::= *... |* alloc(⌧ ) *|* ⇤*e |* null *e* ::= *... |* alloc(⌧ ) *|* ⇤*e |* null We cannot synthesize this type.



### How to Type null?  $\Box$  $\n **F**$  we compute HOW to type fiulire, we allow the independent of  $\mathcal{L}$

*e* ::= *... |* alloc(⌧ ) *|* ⇤*e |* null

Idea: Use an indefinite (polymorphic) type any\* for synthesis One way to capture this is to have a so-called *type subsumption rule* that allows

 $\Gamma\vdash$  null :  $any$   $*$ 

one value (null) which can be compared to a pointer of any definite to a pointer of any definite type. If both<br>If both can be compared to a point of any definite type. If both can be compared to any definite type. If both

- This type can be seen as a temporary placeholder At first glance they might be harmless, but the third rule should raise a red flag: we previously computed the second in our mode and *e* we can give an *e* we can give the can be can  $\overline{\text{mporarv}}$  pl
	- When we constructed the type derivation we could replace any with a 'concrete type' C0 does not have stack-allocated arrays ` *e*<sup>1</sup> : ⌧⇤ ` *e*<sup>2</sup> : ⌧⇤ *e*  $\frac{1}{2}$  **e**  $\frac{1}{2}$  $\epsilon$  *e*<sub>p</sub> : *e*<sub>2</sub> :
- Another view is to say that any\* has exactly one value: null  $\Delta$ pother view is to sev that speak heasexaatly and value A difference with the deregation of the dereference operator  $\frac{1}{2}$  and  $\frac{1}{2}$  and  $\frac{1}{2}$  the version from

## Example: Pointer Equality

### We can compare two pointers using  $p=-q$  if the have the same type

- If p and q both have definite type  $\tau^*$  then  $p == q$  is well-typed
- If p has definite type  $\tau_1$ <sup>\*</sup> and q has definite type  $\tau_2^*$  for different types  $\;\tau_1^{}\;$  and  $\tau_2^{}$ then p==q is rejected
- If p has definite type *τ* \*and q has type any\* then p==q is well typed because we can compare every pointer to null
- If both p and q have type any  $*$  then p==q is well-typed

### Type Rules both have definite type ⌧⇤, we just treat it as well-typed. If one has type ⌧⇤ and the  $\nu$ , we refer the comparison as in the comparison as in the comparison as in the comparison of  $\mu$  $\frac{1}{2}$ we can resolve *any* ⇤, assuming it can only arise for null. Let's consider *pointer equality* figure *pointers. If ypermultiple rather are pointed and*  $\alpha$



#### $\mathbf{e}^1 = \mathbf{e}^2$  $\frac{1}{2}$  : introduced into  $\frac{1}{2}$  : into  $\frac{1}{2$  $\mathsf V$  are not part of the source language and only used internally during type checking. **Equality** The following rule sufficient  $\mathbf{F}\mathbf{q}$  rule sufficient  $\mathbf{r}$  and  $\mathbf{r}$  over  $\mathbf{r}$

 ` *e*<sup>1</sup> == *e*<sup>2</sup> : bool **P**  $\vdash$   $e$  :  $\tau$  $T \vdash e_1 == e_2 : \text{bool}$ where  $\mathcal{L}$  is could estimate any when run, it will be anywhere. Of course, when run, it will be anywhere.  $\frac{1}{\Gamma \vdash e \Rightarrow e \Rightarrow \text{else}}$  $\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau$  $\Gamma \vdash e_1 = e_2 : \mathsf{bool}$ 

## Dynamic semantics of pointers

# Configurations with Heap

- A value of a type  $\tau$   $^*$  is an address that stores a value of type  $\tau$ (or a special address 0)
	- Allocations return fresh (unused) addresses
- Dereferencing retrieves a stored value
- Need heap that maps addresses to values



### Modeling the Heap  $t_0$  $T$

• Addresses are 64 bit words?<br>
The operations and behaviors of the operations and behaviors of the operations and behaviors of the operation Problem: We can run out of memory Frobien. We can full out of memory **fack overflow.** 

exception even the control of memory left. There is still plenty of memory left. There is still plenty of memor<br>The control of memory left. The control of memory left is still plenty of memory left. The control of memory l

We didn't model stack overflow.

- We don't want to specify that programs fail due to memory errors (garbage collection, OS details, …)  $\overline{\mathcal{M}}$  do not model them in the high-level dynamic semantic semantics but all  $\overline{\mathcal{M}}$ we don't want to speeny that programs fair due to memory errors<br>(garbage collection OS details ) that the heap maps natural numbers to values.
- Approach: no out-of-memory errors at the high level -> addresses are natural numbers

 $H : (\mathbb{N} \cup \{\mathsf{next}\}) \to \mathsf{Val}$ 

Evanuation of expressions of the search of the search of the search of the heap of the hea that points to the Special address next free address.



*H* ; *S* ; ⌘ ` *e*<sup>1</sup> *e*<sup>2</sup> B *K* ! *H* ; *S* ; ⌘ ` *e*<sup>1</sup> B (\_ *e*<sup>2</sup> *, K*) Provious runs are lifted.  $\mathcal{A}$  all the prior transition rules leave the heap unchanged. For example,  $\mathcal{A}$ Previous runs are lifted: rovious runs are lift.

Heap is just

 $H$ ;  $S$ ;  $\eta$   $\vdash$   $e_1 \odot e_2 \triangleright K$   $\longrightarrow$   $H$ ;  $S$ ;  $\eta$   $\vdash$   $e_1 \triangleright (\underline{\phantom{e}} \circ e_2, K)$  $H \cdot S \cdot n \vdash A$ 

*Given heap H, stack S, and environment* ⌘*, we evaluate expression e with*

### New rules: The express state a 64-bit word. The expression number and the expression number of  $\mathbb{R}^n$ address 0. Allocation returns a fresh address *a* and maps it to an appropriate de- $A$ l the prior transition rules leave the prior transition rules leave the heap unchanged. For example,  $\mathcal{A}$

 $H$ ;  $S$ ;  $\eta$   $\vdash$  null  $\triangleright$   $K$   $\qquad \longrightarrow$   $H$ ;  $S$ ;  $\eta$   $\vdash$  0  $\triangleright$   $K$  $H \cdot S \cdot m \vdash m \cup \vdash K$  $H : S : \eta \vdash \mathsf{null} \triangleright K \longrightarrow H : S : \eta \vdash 0 \triangleright K$ 

it to an appropriate default value in the new heap. The new heap  $\alpha$ 

Store a default value. *H* ; *S* ; ⌘ ` null B *K* ! *H* ; *S* ; ⌘ ` 0 B *K*

 $H \cdot S \cdot n \vdash \mathsf{alloc}(\tau) \wedge K$  and  $H[\alpha \mapsto \mathsf{def}(\tau) \text{next} \mapsto \alpha + |\tau|] \cdot S \cdot n \vdash \alpha$  $a = H(\text{next})$  and  $\alpha = H(\text{next})$  $\mathbf{f} = \mathbf{f} \cdot \mathbf{f} \cdot \mathbf{f}$ [*a, a* + *|*⌧ *|*) \ dom(*H*) = *{ }, a* 6= 0 The side condition states that none of the allocated addresses are allocated addresses are already in the algorithment of the algorithment  $a = H(\text{next})$ domain of *H*, and that *a* may not be 0 (which would correspond to the null pointer).  $H\,;\,S\,;\,\eta \vdash \mathsf{alloc}(\tau) \rhd K \quad \quad \longrightarrow \quad \quad H[a \mapsto \mathsf{default}(\tau),\mathsf{next} \mapsto a+|\tau|]\,\,;\,S\,;\,\eta \vdash a \rhd K$  $a=H(\mathsf{next})$ 

### **Evaluation Rules: Allocation returns and manufactures** *H* ; *S* ; ⌘ ` alloc(⌧ ) B *K* ! *H*[*a* 7! default(⌧ )] ; *S* ; ⌘ ` *a* B *K*  $\sim$  to an appropriate default value of  $\alpha$  and  $\alpha$  appropriate default value  $\alpha$

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

## implementation, the observer word is always be determined to always be 0 (in whatever word length required to a



 $\rightarrow$  In the implementation you can initialize everything to 0  $\blacksquare$  In the implementation vou can initialize everything to  $\Omega$ This is, of conventions are processed on the processor are processed on the process of the process of the processor and conventions.

Type sizes (x86-64):  $T_{M00}$  cizes (v96-64).

$$
|\text{int}| = 4
$$
\n
$$
|\text{bool}| = 4
$$
\n
$$
|\tau *| = 8
$$
\n
$$
|\tau[]| = 8
$$

# Evaluation Rules: Dereference

### to distinguish "accordinguish" and "on purpose" memory errors. (and "on purpose" memory entrance in purpose" m  $\Gamma$ e us feueresite allows: it allows us to better distinguish stack over  $\Gamma$ to distinguish "accidental" and "on purpose" memory errors.) For two reasons: it allows us to be the better distinguish state over  $\mathcal{L}$  and it allows us to be the better distinguish state over  $\mathcal{L}$

 $H$ ;  $S$ ;  $\eta \vdash *e \triangleright K$   $\longrightarrow$   $H$ ;  $S$ ;  $\eta \vdash e \triangleright (*_-, K)$  $H \cdot S \cdot n \vdash a \wedge (\star \quad K) \longrightarrow H \cdot S \cdot n \vdash H(a) \wedge K \qquad (a \neq 0)$  $\mathcal{L}$  ;  $\mathcal{L}$  ,  $\mathcal{L}$  , If  $\theta$ ,  $\theta$ ,  $\eta$  and  $(\tau$ ,  $\alpha$ ) and a lower corresponding to  $\alpha = 0$ *H* ; *S* ; ⌘ ` ⇤*e* B *K* ! *H* ; *S* ; ⌘ ` *e* B (⇤\_ *, K*)  $H$ ;  $S$ ;  $\eta \vdash a \triangleright (*_-, K) \longrightarrow H$ ;  $S$ ;  $\eta \vdash H(a) \triangleright K \qquad (a \neq 0)$  $H \cdot S \cdot n \vdash a \wedge (k \mid K)$  exception(mem)  $(a = 0)$  $H: \mathcal{C}: \mathbb{R} \mapsto \mathcal{U}$  and  $\mathcal{C}: \mathbb{R}$  $H \cdot S \cdot n \vdash a \wedge (\cdot \cdot \cdot k')$   $H \cdot S \cdot n \vdash H(a) \wedge k'$ *H*  $\frac{1}{2}$  *M*  $\frac{1}{2}$  *<i>M*  $\frac{1}{2}$  *M*  $\frac{1}{2}$  *<i>M*  $\frac{1}{2}$  *M*  $\frac{1}{2}$  *<i>M*  $\frac{1}{2}$   $H$ ;  $S$ ;  $\eta \vdash a \triangleright (*_-, K)$   $\longrightarrow$  exception(mem)  $(a = 0)$ 

for two reasons: it allows us to be two reasons: it allows us to be the better distinguish stack over  $\sim$ 

### Implementing memory exceptions know the size of the size of the data store of the data store at location *at locations*  $\mathbf{A}$ would always be 4 or 8; in C, other sizes would be 4 or 8; in C, other sizes would be possible. In C, other siz<br>The possible experiment of the possible sizes would be possible. In C, other sizes would be possible. In the p In order to implement the correct to implement the lower level of abstraction, we need to be neglected. know the size of the data stored at location *at location*  $\mathbf{r}$  in *H*. Because of our conventions, the size of our conventions, th Implementing memory exceptions

- The purpose CICLICDO incread of CICCECV accomplished via assignment where the destination of dictates the destination of dictates the destination of des • Use signal SIGUSR2 instead of SIGSEGV • Use signal SIGUSR2 instead of SIGSEGV This leaves us with a puzzle: how do we *write* to memory? In C0 (and C) this is
- **Example 1** Ferrer operation. The write operation. The standard *left-handard left-handard left-hand left-hand left-hand left-hand left-hand left-hand left-hand left-hand***</del> <b>***left-hand* accomplished via assignments where the left-hand side dictates the destination of the write operation. These are sometimes called *l-values*, where *l* stands for *left-hand* • Better for debugging: better distinguishable from stack overflow and the write operation. These are sometimes called *l-values*, where *l* stands for *left-hand* "accidental" memory errors

## Assignments: Typing **5 Writing to Heap Destinations 5 Weight to Heap Destinations**

Destinations (or l-values): We define *destinations* (or *l-values*) We define *destinations* (or *l-values*)

 $d ::= x \mid *d$ 

Arrays and structs will add more destinations.

### tination is also a valid expression, so we can just type destinations as expressions. Typing rule: Adding arrays and structs will add more kinds of destinations. Every kind of destimation is also a value. The value of the va<br>The value of the va

 $\overline{D}$   $\overline{D}$   $\overline{D}$   $\overline{D}$   $\overline{D}$   $\overline{D}$  $\begin{array}{c|cccccc} a & . & 1 & & c \\ \hline & & & & & & c \\ \hline & & & & & & & c \\ \hline & & & & & & & c \\ \hline & & & & & & & & c \end{array}$ In the operational semantics we now distinguish variables from other destinations,  $\Gamma \vdash d : \tau \quad \Gamma \vdash e : \tau$  $\Gamma\vdash \mathsf{assign}(d, e) : [\tau']$ Return type of current function.

#### Assignment: Evaluation Rules ` assign(*d, e*) In the operational semantics we now distinguish variables from other destinations, **tion Rule** ation Rules ` assign(*d, e*) ` *d* : ⌧ ` *e* : ⌧ Recall that in typings of statements, ⌧ 0 is the return type of the function that we are ssignment: Evaluation Rules In the operational semantics we now distinguish variables from other destina-destina-destina-destina-destina-

### **Variables:** Variables: stance variables are on the stack (or in registers), which are on the stack (or in registers), which are on the stack (or in  $d$  are on th heap. First, a reminder for assignment if the destination is a variable. The destination is a variable. The des since variables are on the stack (or in registers), while destinations ⇤*d* are on the tions, since variables are on the stack (or in registers), while destinations ⇤*d* are on the heap  $\mathbf{a}$  reminder for assignment if the destination is a variable. The destination is a variable.

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

In the operational semantics we now distinguish variables from other destinations,

In the operational semantics we now distinguish variables from other destinations, we now distinguish variable<br>In the operations, we now distinguish variables from other design variables from our distinguish variables fro

` assign(*d, e*)

#### Memory destinations: If the destination is *not* a variable, we proceed from left to right, first determining the address which is the real memory destination, then evaluate the right-handler then evaluate the right-handler of  $\alpha$ Memory destinations:<br>
Memory destinations: the address which is the real memory destination, then evaluating the right-hand If the destination is *not* a variable, we proceed from left to right, first determining *H* ; *S* ; ⌘ ` assign(*x, e*) B *K* ! *H* ; *S* ; ⌘ ` *e* B (assign(*x,* \_) *, K*) side.

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

### **Examples Mutable Store L15.5 Mutable Store L15.5**

### What happens if we evaluate the following statements? **Detail:** We evaluate the following state is if we evaluate the following stater

Based on the rules above, what should happen in the following code fragments.  $\text{int} * p = \text{NULL}$ ;  $*p = 1/0;$  $int*$  p = NULL:  $kp = 1/0$ : Arithmetic exception.

Based on the rules above, what should happen in the following code fragments.

$$
int** p = NULL; \t Memory**p = 1/0; \t exception.
$$

exception, which is the outcome of the outcome of the execution, which is the execution of the execution. The e<br>The execution of the exec

# Arrays

### Arrays: Typing<br> **Arrays: Typing** Arrays are in many ways similar to pointers, but there are no null arrays. We'll where the outcome of the execution.

Types, expressions, destinations: discusse discusses definitions of the since the single section of the single single single since the single since the single typing rules are more straightforward. discuss default arrays destinations: For now, the since the since the single since the single since the single since the single single since the single single since the single single single single single single single sing **6 Arraysing** 

$$
\tau ::= \dots | \tau[ ]
$$
\nThere are no  
\n
$$
e ::= \dots | \text{alloc\_array}(\tau, e) | e_1[e_2]
$$
\n"null" arrays.  
\n
$$
d ::= \dots | d[e]
$$
\nHowever,  
\nthere are

Arrays are in many ways similar to pointers, but there are no null arrays. We'll

There are no "null" arrays.

 $\mathbf{t}$  :  $\$ e<br>defa unere are<br>default arrays. *e all* carry or *electric array* there are

### Type rules: Type rules for all othat semantics for all othat semantics for memory and memo  $\mathbf{F}$ ,  $\mathbf{F}$ ,  $\mathbf{F}$ ,  $\mathbf{F}$  array  $\mathbf{F}$ The dynamic semantics for all other allocation obtains a fresh segment of memory and memory and memory and memory and  $\alpha$ Type rules: *e* ::= *... |* alloc array(⌧*, e*) *| e*1[*e*2] *d* ::= *... | d*[*e*] Type rules:

$$
\frac{\Gamma \vdash e_1 : \tau[] \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1[e_2] : \tau} \qquad \qquad \frac{\Gamma \vdash e : \mathsf{int}}{\Gamma \vdash \mathsf{alloc\_array}(\tau, e) : \tau[]}
$$

 $\frac{1}{\prod_{i=1}^{n} a_i}$   $\frac{2}{\prod_{i=1}^{n} b_i}$   $\frac{1}{\prod_{i=1}^{n} b_i}$   $\frac{1}{\prod_{i=1}^{n} b_i}$  $\Gamma \vdash e_1[e_2] : \tau$   $\Gamma \vdash \mathsf{alloc\_array}(\tau, e) : \tau[$  $\Gamma \vdash e : \mathsf{int}$  $\Gamma \vdash$  alloc\_array $(\tau, e) : \tau[$  ]

#### Array Evaluation: Access *Array*  $\overline{ACC}$ ess *a*  $\mathcal{F}(\mathcal{A})$  ;  $\mathbb{E}_{\mathcal{A}}$  ob  $\mathbb{E}_{\mathcal{A}}$  of  $\mathbb{E}_{\mathcal{A}}$  and  $\mathbb{E}_{\mathcal{A}}$  a  $H$  is a set of the B (*n* B  $\alpha$  and  $\alpha$  are  $\alpha$  in B  $\alpha$  and  $\alpha$  are  $\alpha$ Array Fyaluation: Access *H* ; *S* ; ⌘ ` *n* B (alloc array(⌧*,* \_) *, K*) ! *H*<sup>0</sup> ; *S* ; ⌘ ` *a* B *K* (*n* 0) individuation: Access all *n* elements of the array Evaluation: Access and the default value of ` alloc array(⌧*, e*) : ⌧ [ ] ` *e*1[*e*2] : ⌧ Array Evaluation: Access

initializes all *n* elements of the array with the default value of type ⌧ .

$$
H; S; \eta \vdash e_{1}[e_{2}] \rhd K \longrightarrow H; S; \eta \vdash e_{1} \rhd (\lfloor e_{2} \rfloor, K)
$$
\n
$$
H; S; \eta \vdash a \rhd (\lfloor e_{2} \rfloor, K) \longrightarrow H; S; \eta \vdash e_{2} \rhd (a \lfloor \rfloor, K)
$$
\nNeed types.

\n
$$
H; S; \eta \vdash i \rhd (a \lfloor \rfloor, K) \longrightarrow H; S; \eta \vdash H(a + i|\tau|) \rhd K
$$
\n
$$
a \neq 0, 0 \leq i < \text{length}(a), a : \tau[]
$$
\nNeed array sizes.

\n
$$
H; S; \eta \vdash i \rhd (a \lfloor \rfloor, K) \longrightarrow \text{exception}(mem)
$$
\n
$$
a = 0 \text{ or } i < 0 \text{ or } i \geq \text{length}(a)
$$

 $A$ ray access evaluates from left to right and then computes then computes then computes then computes the correct memory  $\mathcal{A}$ 

*H* ; *S* ; ⌘ ` alloc array(⌧*, e*) B *K* ! *H* ; *S* ; ⌘ ` *e* B (alloc array(⌧*,* \_) *, K*)

[*a, a* + *n|*⌧ *|*) \ dom(*H*) = *{ }, a* 6= 0

[*a, a* + *n|*⌧ *|*) \ dom(*H*) = *{ }, a* 6= 0

*H* ; *S* ; ⌘ ` *n* B (alloc array(⌧*,* \_) *, K*) ! exception(mem) (*n <* 0)

# Arrays: Implementation

## Types?

- We know type  $τ[]$  of destination e1 at compile time
- ➡ Just select the right constant when generating code
- In the dynamic semantics: assume  $e_1[e_2]$  has been elaborated to

$$
e_1\{\tau\}[e_2] \qquad \text{if} \qquad e_1:\tau
$$

## Lengths?

- Not known at compile time
- In alloc\_array(τ,e), e can be an arbitrary expression
- ➡Need to store array length

### Storing Array Length **Detail: Storing the Array Length** This could be layed out as follows, where *a* is the address of the array *A* with One possibility is to allocate a few additional bytes to store the length of the array. elements of the USA And Storing And Storing And St

Alternative 1: Add length at the front, array address points to the start ive 1: Ac 1: Add length at the front, array address points to the



### Alternative 2: Array address points to the first element Alternative 2: Array address points to the first element



- Simplifies address arithmetic *<sup>n</sup>*  $\frac{1}{2}$ @  $\alpha$  *A*ddress arithmetic
- Allows to pass pointers to C (which wouldn't care about length info) **a** a a a above the pace pointers to C (which wouldn't care about length info) Allows to pass pointers to C (which wouldn't care about length into)

#### Updated Rules for Array Access I Indated Rules for Array Access O puditud i idios ion Antay Access ated J L @ and Rules for Array Access pointer directly to C (which would not care about the length information to the

that way.

*<sup>n</sup>*

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

The reason we locate the length *n* at *a* 8 and not *a* 4 is so that *a* itself will be

@@ *<sup>A</sup>*[0] *··· <sup>A</sup>*[*<sup>n</sup>* 1]

*a*8 *a*4 *a a*+(*n*1)*|*⌧*|*

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

$$
H; S; \eta \vdash i \rhd (a\{\tau\}\llbracket \cdot \rrbracket, K) \longrightarrow H; S; \eta \vdash H(a+i|\tau|) \rhd K
$$
  

$$
a \neq 0, 0 \leq i < H(a-8)
$$

 $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$   $\frac{1}{2}$ 

 $H$ ;  $S$ ;  $\eta$   $\vdash$  *i*  $\triangleright$  (a{ $\tau$ }[\_],  $K$ )  $\longrightarrow$  exception(mem)  $H$ ;  $S$ ;  $\eta \vdash i \rhd (a\{\tau\}[\_], K)$   $\longrightarrow$  exception(mem)

*r* ention (mem)  $a = 0 \text{ or } i < 0 \text{ or } i > H(a - 8)$ *a* = *H*(next) *a*0 = *a* + 8 *a* = 0 or *i <* 0 or *i H*(*a* 8) *H* ; *S* ; ⌘ ` alloc array(⌧*, e*) B *K* ! *H* ; *S* ; ⌘ ` *e* B (alloc array(⌧*,* \_) *, K*) *a* 6= 0*,* 0 *i<H*(*a* 8)  $a = 0$  or  $i < 0$  or  $i \geq H(a - 8)$ 

## Array Access: Code Generation

The code pattern for  $e_1\{\tau\}[e_2]$  and  $|\tau| = k$  could be like this:

$$
cogen(e1, a) (a new)\n
$$
cogen(e2, i) (i new)\n
$$
a1 \leftarrow a - 8\n
$$
t2 \leftarrow M[a1]\nif (i < 0) goto error\nif (i  $\geq t2$ ) goto error  
\n
$$
a3 \leftarrow i * \$k\n
$$
a4 \leftarrow a + a3\n
$$
t5 \leftarrow M[a4]
$$
\n(4)
$$
$$
$$
$$
$$
$$

### Array Evaluation: Allocation  $\overline{a}$ <sup>+</sup> $\overline{a}$ valuation. Allow  $\overline{\phantom{a}}$ Array Evaluation: Allocation

typing rules are more straightforward. The more straightforward is a straightforward of the straightforward.

 $H$ ;  $S$ ;  $\eta$   $\vdash$  alloc\_array( $\tau$ ,  $e$ )  $\vartriangleright$   $K$   $\qquad \longrightarrow$   $H$ ;  $S$ ;  $\eta$   $\vdash$   $e$   $\vartriangleright$  (alloc\_array( $\tau$ ,  $\_,$ ),  $K$ )  $F(e) \triangleright K$   $\longrightarrow$   $H: S: n$  $\alpha$   $\beta$   $\beta$   $\beta$   $\beta$ *a* 6= 0*,* 0 *i<H*(*a* 8)  $S$ ;  $\eta$   $\vdash$  alloc\_array( $\tau$ ,  $e$ )  $\triangleright$   $K$   $\qquad \longrightarrow$   $\qquad$   $H$ ;  $S$ ;  $\eta$   $\vdash$   $e$  (

The dynamic semantics for all other dynamic semantics for all other dynamic semantics for  $\alpha$  fresh segment of  $\alpha$ 

*H* ; *S* ; ⌘ ` *a* B (\_*{*⌧*}*[*e*2] *, K*) ! *H* ; *S* ; ⌘ ` *e*<sup>2</sup> B (*a{*⌧*}*[\_] *, K*)

$$
H; S; \eta \vdash n \rhd (\text{alloc\_array}(\tau, \_) , K) \longrightarrow H' ; S; \eta \vdash a' \rhd K \quad (n \ge 0)
$$
  
\n
$$
a = H(\text{next}) \quad a' = a + 8
$$
  
\n
$$
H' = H[a \mapsto n, a' + 0|\tau| \mapsto \text{default}(\tau), \dots, a' + (n - 1)|\tau| \mapsto \text{default}(\tau), \text{next} \mapsto a' + n|\tau|]
$$

 $\longrightarrow$  exception(mem)  $(n < 0)$ *H* ; *S* ; ⌘ ` *i* B (*a*[\_] *, K*) ! exception(mem) *a* = 0 or *i <* 0 or *i* length(*a*)  $H$  ;  $S$  ;  $\eta \vdash n \rhd (\text{alloc\_array}(\tau, \_)$  ,  $K)$ 

 $\bigcup$  length(a) = H(a-8) a congram, we somewhat the find out it somewhere  $\alpha$ properties of two complements of two complements of two complements in the two comparisons in the two comparisons in

$$
H\mathbin{;} S\mathbin{;} \eta\vdash \mathrm{assign}(d\{\tau\}[e_2],e_3)\blacktriangleright K\hspace{1cm}\longrightarrow\hspace{1cm} H\mathbin{;} S\mathbin{;} \eta\vdash d\mathbin{\vartriangleright} \big(\mathrm{assign}(\_ \{\tau\}[e_2],e_3)\mathbin{,} K)
$$

bytes. We have written \$*k* to indicate that this is an immediate operand (that is, a

Executing assignments with the new destinations is quite similar to reading.

to avoid some intermediate computation such as *a*<sup>1</sup> or *a*4. Also, we can exploit

$$
H\mathbin{;} S\mathbin{;} \eta\vdash a\mathbin{\vartriangleright} \big(\textnormal{assign}(\_,\{\tau\}[e_2],e_3)\mathbin{;} K\big)\qquad\mathop{\longrightarrow}\qquad H\mathbin{;} S\mathbin{;} \eta\vdash e_2\mathbin{\vartriangleright} \big(\textnormal{assign}(a\{\tau\}[\_,e_3)\mathbin{;} K\big)
$$

$$
\begin{array}{lcl} H\,;\,S\,;\,\eta \vdash i \rhd (\mathsf{assign}(a\{\tau\}\llbracket\_], e_3)\,\,,\,K) & \longrightarrow & H\,;\,S\,;\,\eta \vdash e_3 \rhd (\mathsf{assign}(a + i|\tau|,\_\_)\,\,,\,K)\\ & & a \neq 0, 0 \leq i < \mathsf{length}(a) \end{array}
$$

*H* ; *S* ; ⌘ ` assign(*d*1[*e*2]*, e*3) I *K* ! *H* ; *S* ; ⌘ ` *e*<sup>1</sup> B (assign(\_[*e*2]*, e*3) *, K*)

*H* ; *S* ; ⌘ ` *i* B (assign(*a{*⌧*}*[\_]*, e*3) *, K*) ! *H* ; *S* ; ⌘ ` *e*<sup>3</sup> B (assign(*a* + *i|*⌧ *|,* \_) *, K*)

 $S$  ;  $\eta \vdash i \rhd (\textsf{assign}(a\{\tau\}[\_], e_3)$  ,  $K)$   $\longrightarrow$  exception(mem)  $a = 0$  or  $i < 0$  or  $i \geq$  length $(a)$ **7 Values of Array Type** Here, we have written length(*a*) for *H*(*a* 8). *a* = 0 or *i <* 0 or *i* length(*a*)  $H$ ;  $S$ ;  $\eta \vdash i \rhd (\mathsf{assign}(a\{\tau\}[\_], e_3)$ ,  $K)$   $\longrightarrow$  exception(mem) *a* 6= 0*,* 0 *i <* length(*a*)

*a* = 0 or *i <* 0 or *i* length(*a*)  $H\,;\,S\,;\,\eta \vdash c \rhd (\mathsf{assign}(b,\_) \;,\, K) \qquad \qquad \longrightarrow \qquad H[b \mapsto c] \;;\,S\,;\,\eta \vdash \mathsf{nop} \blacktriangleright K$ 0, which represents an array of size 0. We can never legally access any element of  $H \cdot S \cdot n \vdash c \triangleright (a \cdot s \cdot n) \quad K) \qquad \longrightarrow \qquad H[b \mapsto c] \cdot S \cdot n \vdash \text{non} \blacktriangleright K$ false), and for pointers it is 0 (which represents numbered in a representation of arrays is also in a represent<br>In also in a rate of a representation of a rate of arrays is also in a rate of a rate of a rate of a rate of  $11, 0, 11$  CD (assign(0, -),  $H$ ;  $S$ ;  $\eta \vdash c \rhd (\mathsf{assign}(b, \_)$ ,  $K)$ 

# Default Values of Array Type

## We also need a default value for array types

- We will just use 0 as the default value again
- It represents an array of length 0
- We can never legally access an array element in the default array
- Warning: Arrays can be compared with equality
- Make sure that alloc\_array(t,0) returns a fresh address different from 0
- If arrays have address a=0 then you should not access M[a-8]

## Compound Assignment Operators

- We translate  $x == e$  to  $x = x + e$
- We cannot translate  $d1[e2] += e3$  to  $d1[e2] = d1[e2] + e3$

Effects of e2 and d1 would be evaluated twice.