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

Static semantics of pointers

Static Semantics of Pointers

Static Semantics of Pointers

Extend types with pointer types:

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

Static Semantics of Pointers

Extend types with pointer types:

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

Extend expressions with allocations, dereference, and null pointers:

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

Static Semantics of Pointers

Extend types with pointer types:

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

Extend expressions with allocations, dereference, and null pointers:

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

We add the following typing rules for expressions:

$$\frac{}{\Gamma \vdash \mathsf{alloc}(\tau) : \tau*} \qquad \frac{\Gamma \vdash e : \tau*}{\Gamma \vdash *e : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \tau*}$$

Static Semantics of Pointers

Extend types with pointer types:

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

Extend expressions with allocations, dereference, and null pointers:

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

We add the following typing rules for expressions:

$$\frac{}{\Gamma \vdash \mathsf{alloc}(\tau) : \tau*} \qquad \frac{\Gamma \vdash e : \tau*}{\Gamma \vdash *e : \tau} \qquad \frac{}{\Gamma \vdash \mathsf{null} : \tau*}$$

We cannot synthesize this type.

How to Type null?

Idea: Use an indefinite (polymorphic) type `any*` for synthesis

How to Type null?

Idea: Use an indefinite (polymorphic) type `any*` for synthesis

$$\frac{}{\Gamma \vdash \textsf{null} : \mathit{any}\, *}$$

How to Type null?

Idea: Use an indefinite (polymorphic) type `any*` for synthesis

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

- This type can be seen as a temporary placeholder
- When we constructed the type derivation we could replace any with a ‘concrete type’
- Another view is to say that `any*` has exactly one value: null

Example: Pointer Equality

We can compare two pointers using `p==q` if the have the same type

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

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 $\tau_1 *$
  and q has definite type $\tau_2 *$ for different types τ_1 and τ_2
  then `p==q` is rejected

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 τ * then `p==q` is well-typed

- If p has definite type τ_1 *
  and q has definite type τ_2 * for different types τ_1 and τ_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

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 τ_1*
  and q has definite type τ_2* for different types τ_1 and τ_2
  then `p==q` is rejected

- If `p` has definite type $\tau*$
  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

Dereference and type instantiation

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

Equality

$$\frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{==} e_2 : \mathsf{bool}}$$

Type Rules

Dereference and type instantiation

Cannot dereference a Null pointer.

$$\frac{\Gamma \vdash e : any\,*}{\Gamma \vdash e : \tau *} \qquad \frac{\Gamma \vdash e : \tau * \quad \Gamma \nvdash e : any\,*}{\Gamma \vdash *e : \tau}$$

Equality

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

Dynamic semantics of pointers

Configurations with Heap

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

➡ Need **heap** that maps addresses to values

Configurations with Heap

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

➡ Need **heap** that maps addresses to values

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

Configurations with Heap

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

➡ Need **heap** that maps addresses to values

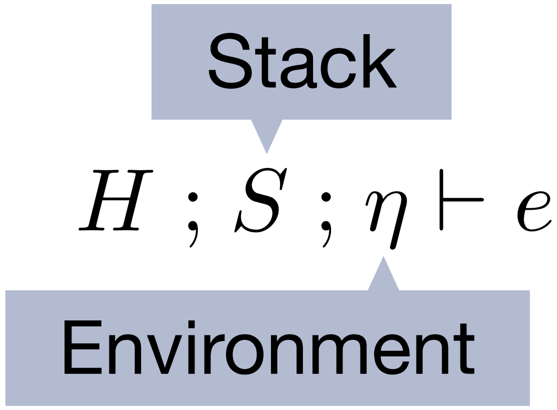
Stack

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

Configurations with Heap

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

➡ Need **heap** that maps addresses to values



Stack

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

Environment

Configurations with Heap

- A value of a type τ^* is an address that stores a value of type τ (or a special address 0)
- Allocations return fresh (unused) addresses
- Dereferencing retrieves a stored value

➡ Need **heap** that maps addresses to values

Stack

Continuation

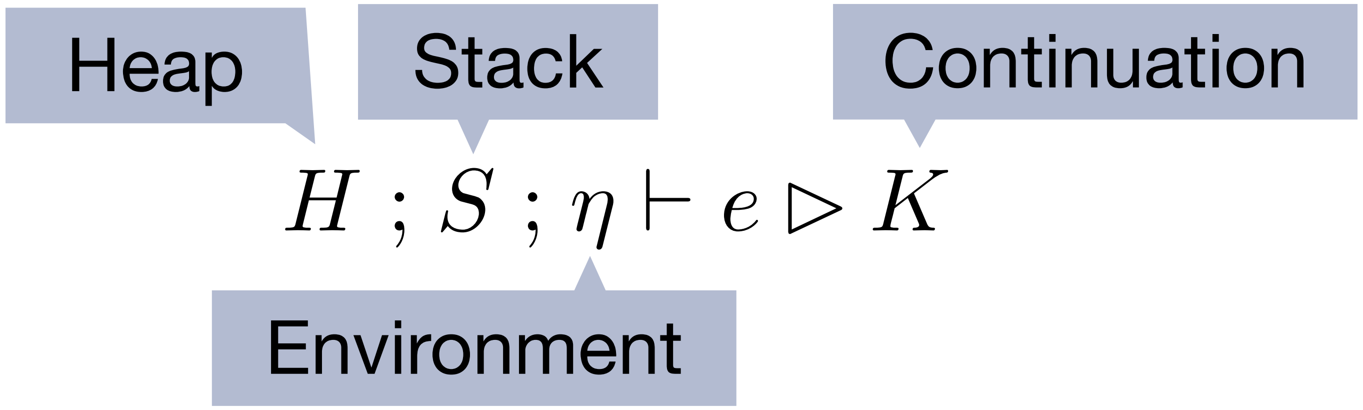
$$H\ ;\ S\ ;\ \eta \vdash e \rhd K$$

Environment

Configurations with Heap

- A value of a type τ^* is an address that stores a value of type τ (or a special address 0)
- Allocations return fresh (unused) addresses
- Dereferencing retrieves a stored value

➡ Need **heap** that maps addresses to values



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

Modeling the Heap

- Addresses are 64 bit words?
  Problem: We can run out of memory

- We don’t want to specify that programs fail due to memory errors (garbage collection, OS details, …)

- Approach: no out-of-memory errors at the high level
  -> **addresses are natural numbers**

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

Modeling the Heap

- Addresses are 64 bit words?
  Problem: We can run out of memory

- We don't want to specify that programs fail due to memory errors (garbage collection, OS details, …)

- Approach: no out-of-memory errors at the high level
  -> **addresses are natural numbers**

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

Special address that points to the next free address.

Modeling the Heap

- Addresses are 64 bit words?
  Problem: We can run out of memory

We didn't model stack overflow.

- We don't want to specify that programs fail due to memory errors (garbage collection, OS details, …)

- Approach: no out-of-memory errors at the high level
  -> **addresses are natural numbers**

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

Special address that points to the next free address.

Evaluation Rules I

Evaluation Rules I

Previous runs are lifted:

$$H\ ;\ S\ ;\ \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e_1 \rhd (\_ \odot e_2\ ,\ K)$$

Evaluation Rules I

Heap is just passed through.

Previous runs are lifted:

$$H\ ;\ S\ ;\ \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e_1 \rhd (\_ \odot e_2\ ,\ K)$$

Evaluation Rules I

Heap is just passed through.

Previous runs are lifted:

$$H\ ;\ S\ ;\ \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e_1 \rhd (\_ \odot e_2\ ,\ K)$$

New rules:

Evaluation Rules I

Heap is just passed through.

Previous runs are lifted:

$$H \; ; S \; ; \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash e_1 \rhd (\_ \odot e_2 \; , K)$$

New rules:

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

Evaluation Rules I

Heap is just passed through.

Previous runs are lifted:

$$H \; ; \; S \; ; \; \eta \vdash e_1 \odot e_2 \rhd K \quad \longrightarrow \quad H \; ; \; S \; ; \; \eta \vdash e_1 \rhd (\_ \odot e_2 \; , \; K)$$

New rules:

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

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

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

Evaluation Rules I

Heap is just passed through.

Previous runs are lifted:

$$H \; ; \; S \; ; \; \eta \vdash e_1 \odot e_2 \rhd K \qquad \longrightarrow \qquad H \; ; \; S \; ; \; \eta \vdash e_1 \rhd (\_ \odot e_2 \; , \; K)$$

New rules:

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

Store a default value.

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

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

Evaluation Rules: Allocation

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

Default values

default(bool) = false

default(int) = 0

default($\tau*$) = null

Evaluation Rules: Allocation

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

Default values

default(bool) = false

default(int) = 0

default(τ *) = null

➡ In the implementation you can initialize everything to 0

Evaluation Rules: Allocation

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

Default values

default(bool) = false   default(int) = 0   default(τ *) = null

➡ In the implementation you can initialize everything to 0

Type sizes (x86-64):

$$\begin{array}{lcl} |\mathsf{int}| & = & 4 \\ |\mathsf{bool}| & = & 4 \\ |\tau *| & = & 8 \\ |\tau[\,]| & = & 8 \end{array}$$

Evaluation Rules: Dereference

Dereferencing

Evaluation Rules: Dereference

Dereferencing

$$H\ ;\ S\ ;\ \eta \vdash *e \triangleright K \qquad\qquad \longrightarrow \qquad\qquad H\ ;\ S\ ;\ \eta \vdash e \triangleright (*\_\ ,\ K)$$

Evaluation Rules: Dereference

Dereferencing

$$H\ ;\ S\ ;\ \eta \vdash *e \rhd K \qquad \longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash e \rhd (*\_\ ,\ K)$$

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

Evaluation Rules: Dereference

Dereferencing

$$H\ ;\, S\ ;\, \eta \vdash *e \rhd K \quad \longrightarrow \quad H\ ;\, S\ ;\, \eta \vdash e \rhd (*\_\ ,\, K)$$

$$H\ ;\, S\ ;\, \eta \vdash a \rhd (*\_\ ,\, K) \quad \longrightarrow \quad H\ ;\, S\ ;\, \eta \vdash H(a) \rhd K \qquad (a \neq 0)$$

$$H\ ;\, S\ ;\, \eta \vdash a \rhd (*\_\ ,\, K) \quad \longrightarrow \quad \textsf{exception}(\textsf{mem}) \qquad (a = 0)$$

Evaluation Rules: Dereference

Dereferencing

$$H \; ; S \; ; \eta \vdash *e \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash e \rhd (*\_ \; , K)$$

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

$$H \; ; S \; ; \eta \vdash a \rhd (*\_ \; , K) \qquad \longrightarrow \qquad \mathsf{exception}(\mathsf{mem}) \qquad (a = 0)$$

Implementing memory exceptions

- Use signal SIGUSR2 instead of SIGSEGV
- Better for debugging: better distinguishable from stack overflow and “accidental” memory errors

Assignments: Typing

Destinations (or l-values):

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

Assignments: Typing

Destinations (or l-values):

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

Arrays and structs will add more destinations.

Assignments: Typing

Destinations (or l-values):

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

Arrays and structs will add more destinations.

Typing rule:

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

Assignments: Typing

Destinations (or l-values):

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

Arrays and structs will add more destinations.

Typing rule:

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

Return type of current function.

Assignment: Evaluation Rules

Variables:

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

Assignment: Evaluation Rules

Variables:

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

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

Assignment: Evaluation Rules

Variables:

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

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

Memory destinations:

Assignment: Evaluation Rules

Variables:

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

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

Memory destinations:

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

Assignment: Evaluation Rules

Variables:

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

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

Memory destinations:

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

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

Assignment: Evaluation Rules

Variables:

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

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

Memory destinations:

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

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

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

Assignment: Evaluation Rules

Variables:

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

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

Memory destinations:

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

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

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

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

Examples

What happens if we evaluate the following statements?

Examples

What happens if we evaluate the following statements?

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

Examples

What happens if we evaluate the following statements?

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

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

Examples

What happens if we evaluate the following statements?

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

Arithmetic exception.

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

Examples

What happens if we evaluate the following statements?

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

Arithmetic exception.

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

Memory exception.

Arrays

Arrays: Typing

Types, expressions, destinations:

Arrays: Typing

Types, expressions, destinations:

$$\tau \quad ::= \quad \ldots \mid \tau[\,]$$

Arrays: Typing

Types, expressions, destinations:

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

Arrays: Typing

Types, expressions, destinations:

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

Arrays: Typing

Types, expressions, destinations:

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

Type rules:

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

Arrays: Typing

Types, expressions, destinations:

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

There are no “null” arrays.

Type rules:

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

Arrays: Typing

Types, expressions, destinations:

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

There are no "null" arrays.

However, there are default arrays.

Type rules:

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

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

Array Evaluation: Access

Array Evaluation: Access

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

Array Evaluation: Access

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

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

Array Evaluation: Access

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

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

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

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

Array Evaluation: Access

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

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

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

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

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

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

Array Evaluation: Access

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

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

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

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

Need array sizes.

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

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

Array Evaluation: Access

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

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

Need types.

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

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

Need array sizes.

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

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

Arrays: Implementation

Types?

Arrays: Implementation

Types?

- We know type $\tau[]$ 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

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

Arrays: Implementation

Types?

- We know type $\tau[]$ 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?

Arrays: Implementation

Types?

- We know type $\tau[]$ 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

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

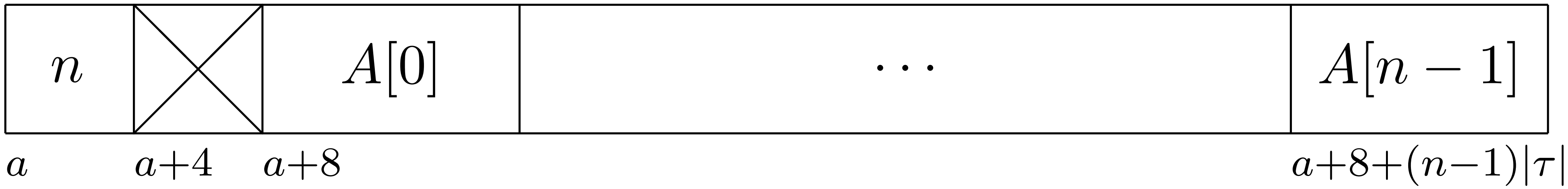
Lengths?

- Not known at compile time
- In `alloc_array(`τ`,e),` `e` can be an arbitrary expression

➡Need to store array length

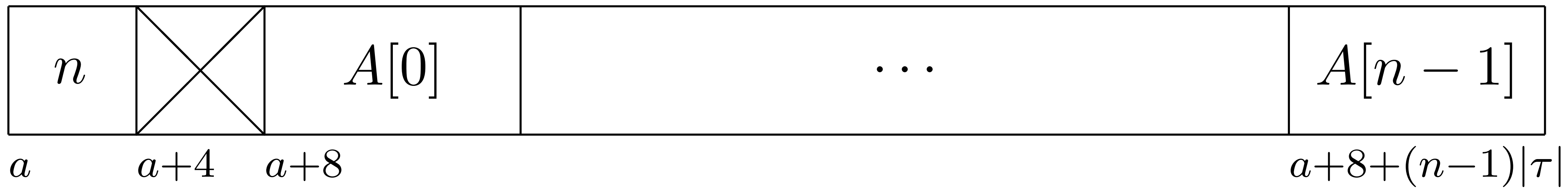
Storing Array Length

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

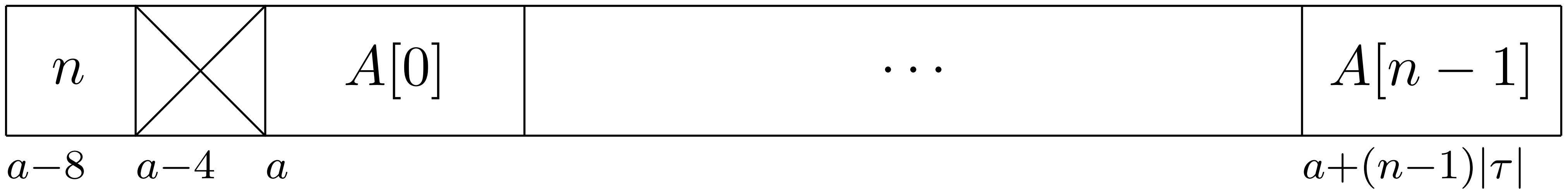


Storing Array Length

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

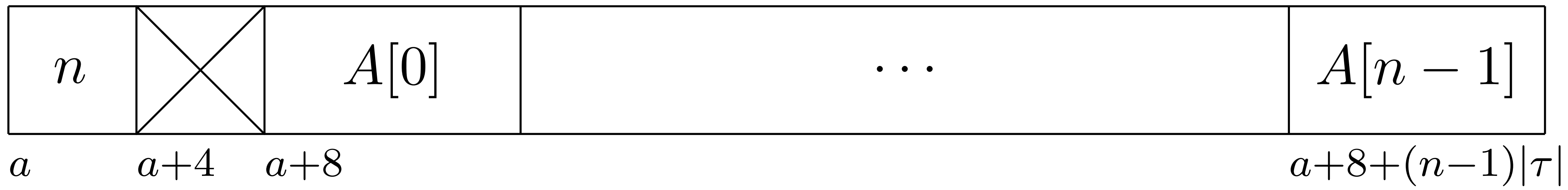


Alternative 2: Array address points to the first element

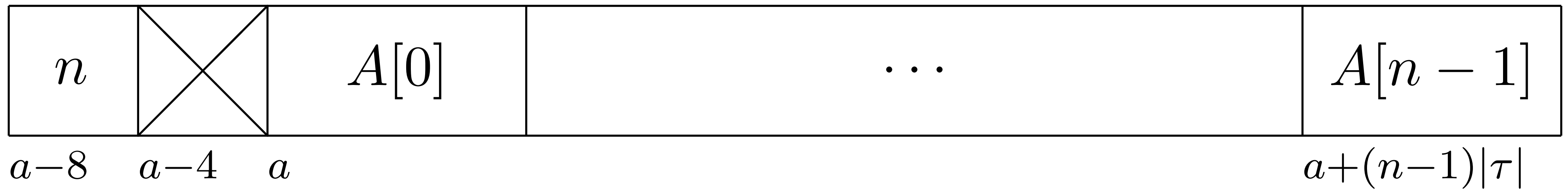


Storing Array Length

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



Alternative 2: Array address points to the first element



- Simplifies address arithmetic
- Allows to pass pointers to C (which wouldn't care about length info)

Updated Rules for Array Access

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

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

$$H\ ;S\ ;\eta \vdash i \rhd (a\{\tau\}[\_]\ ,K) \quad\longrightarrow\quad \begin{array}{l} H\ ;S\ ;\eta \vdash H(a+i|\tau|) \rhd K \\ a \neq 0, 0 \leq i < H(a-8) \end{array}$$

$$H\ ;S\ ;\eta \vdash i \rhd (a\{\tau\}[\_]\ ,K) \quad\longrightarrow\quad \begin{array}{l} \textsf{exception(mem)} \\ a = 0 \text{ or } i < 0 \text{ or } i \geq H(a-8) \end{array}$$

Array Access: Code Generation

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

```
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
$a_3 \leftarrow i * \$k$
$a_4 \leftarrow a + a_3$
$t_5 \leftarrow M[a_4]$
```

Array Evaluation: Allocation

Array Evaluation: Allocation

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

Array Evaluation: Allocation

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

$$H\ ;\ S\ ;\ \eta \vdash n \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K) \quad \longrightarrow \quad H'\ ;\ S\ ;\ \eta \vdash a' \rhd K \qquad (n \geq 0)$$

$$a = H(\mathsf{next}) \qquad a' = a + 8$$

$$H' = H[a' + 0|\tau| \mapsto \mathsf{default}(\tau), \ldots, a' + (n-1)|\tau| \mapsto \mathsf{default}(\tau), \mathsf{next} \mapsto a' + n|\tau|]$$

Array Evaluation: Allocation

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

$$H\ ;\ S\ ;\ \eta \vdash n \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K) \qquad \longrightarrow \qquad H'\ ;\ S\ ;\ \eta \vdash a' \rhd K \qquad (n \geq 0)$$

$$a = H(\mathsf{next}) \qquad a' = a + 8$$

$$H' = H[a' + 0|\tau| \mapsto \mathsf{default}(\tau), \ldots, a' + (n-1)|\tau| \mapsto \mathsf{default}(\tau), \mathsf{next} \mapsto a' + n|\tau|]$$

$$H\ ;\ S\ ;\ \eta \vdash n \rhd (\mathsf{alloc\_array}(\tau, \_)\ ,\ K) \qquad \longrightarrow \qquad \mathsf{exception}(\mathsf{mem}) \qquad (n < 0)$$

Array Evaluation: Assignment

Array Evaluation: Assignment

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

Array Evaluation: Assignment

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

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

Array Evaluation: Assignment

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

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

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

Array Evaluation: Assignment

length(a) = H(a-8)

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

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

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

$$a \neq 0, 0 \leq i < \mathsf{length}(a)$$

Array Evaluation: Assignment

length(a) = H(a-8)

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

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

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

$$a \neq 0, 0 \leq i < \mathsf{length}(a)$$

$$H \; ; \; S \; ; \; \eta \vdash i \rhd (\mathsf{assign}(a\{\tau\}[\_], e_3)\ ,\ K) \quad \longrightarrow \quad \mathsf{exception}(\mathsf{mem})$$

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

Array Evaluation: Assignment

length(a) = H(a-8)

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

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

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

$$a \neq 0, 0 \leq i < \mathsf{length}(a)$$

$$H\ ;\ S\ ;\ \eta \vdash i \rhd (\mathsf{assign}(a\{\tau\}[\_], e_3)\ ,\ K) \quad\longrightarrow\quad \mathsf{exception}(\mathsf{mem})$$

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

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

Default Values of Array Type

We also need a default value for array types

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

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.