15-411: Structs

Jan Hoffmann

Struct Declarations and Definitions

Declaring structs:

$$\textsf{struct}\ s;$$

Defining structs:

$$\textsf{struct}\ s\ \{\tau_1\ f_1; \ldots \tau_n\ f_n;\};$$

Struct Declarations and Definitions

Declaring structs:

$$\text{struct } s;$$

Type

Defining structs:

$$\text{struct } s \; \{\tau_1 \; f_1; \ldots \tau_n \; f_n; \};$$

Struct Declarations and Definitions

Declaring structs:

$$\text{struct } s;$$

Type

Defining structs:

Field

$$\text{struct } s \; \{\tau_1 \; f_1; \ldots \tau_n \; f_n; \};$$

Struct Declarations and Definitions

Declaring structs:

$$\text{struct } s;$$

Type

Defining structs:

Field

$$\text{struct } s\ \{\tau_1\ f_1; \ldots \tau_n\ f_n;\};$$

Type

Struct Declarations and Definitions

Declaring structs:

$$\texttt{struct } s;$$

Type

Defining structs:

Field

$$\texttt{struct } s\ \{\tau_1\ f_1; \ldots \tau_n\ f_n;\};$$

Type

During type derivation we write

$$s.f_i \ : \ \tau_i$$

Small and Large Types

- Arrays are represented with pointers (but cannot be dereferenced)
  -> they can be compared and stored in registers
- Structs are usually also pointers but they can be dereferenced (Why?)
- Structs are large types that do not fit registers

Small and Large Types

- Arrays are represented with pointers (but cannot be dereferenced)
  -> they can be compared and stored in registers
- Structs are usually also pointers but they can be dereferenced (Why?)
- Structs are large types that do not fit registers

Small types:

int, bool, $\tau*$, $\tau[\,]$

Large types:

struct s

Restrictions on Large Types

Restrictions on Large Types

- **Local variables,function parameters,and return values** must have small type

Restrictions on Large Types

- **Local variables,function parameters,and return values** must have small type

- **Left- and right-hand sides of assignments** must have small type

Restrictions on Large Types

- **Local variables,function parameters,and return values** must have small type

- **Left- and right-hand sides of assignments** must have small type

- **Conditional expressions** must have small type

Restrictions on Large Types

- **Local variables,function parameters,and return values** must have small type
- **Left- and right-hand sides of assignments** must have small type
- **Conditional expressions** must have small type
- **Equality and disequality** must compare expressions of small type

Restrictions on Large Types

- **Local variables,function parameters,and return values** must have small type
- **Left- and right-hand sides of assignments** must have small type
- **Conditional expressions** must have small type
- **Equality and disequality** must compare expressions of small type
- **Expressions** used as statements must have small type

Restrictions on Large Types

In C0

- **Local variables,function parameters,and return values** must have small type
- **Left- and right-hand sides of assignments** must have small type
- **Conditional expressions** must have small type
- **Equality and disequality** must compare expressions of small type
- **Expressions** used as statements must have small type

Static Semantics

Semantic Rules For Structs

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)
- The same field names can be used in different struct definitions

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)
- The same field names can be used in different struct definitions
- In a given struct definition, all field names must be distinct

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)
- The same field names can be used in different struct definitions
- In a given struct definition, all field names must be distinct
- A struct may be defined at most once

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)
- The same field names can be used in different struct definitions
- In a given struct definition, all field names must be distinct
- A struct may be defined at most once
- Types struct s that have not (yet) been defined may be referenced as long as their size is irrelevant; size is relevant for `alloc(struct s)`
  `alloc_array(struct s,e)`
  structs if structs are types of fields

Semantic Rules For Structs

- Field names occupy their own namespace: allowed to overlap with variable, function, or type names (but they must be distinct from keywords)
- The same field names can be used in different struct definitions
- In a given struct definition, all field names must be distinct
- A struct may be defined at most once
- Types struct s that have not (yet) been defined may be referenced as long as their size is irrelevant; size is relevant for `alloc(struct s)` `alloc_array(struct s,e)` structs if structs are types of fields
- An occurrence of struct s in a context where its size is irrelevant serves as an implicit declaration of the type struct s. In effect this means that explicit struct declarations are optional (but encouraged as good style)

Expressions and Typing

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \cdots \ \big| \ \textsf{struct}\ s$$

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \dots \mid \mathsf{struct}\ s$$

Extend expressions with field access:

$$\begin{array}{lll} e & ::= & \dots \mid e.f \\ d & ::= & \dots \mid d.f \end{array}$$

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \cdots \ \Big| \ \mathsf{struct}\ s$$

Extend expressions with field access:

$$\begin{array}{lcl} e & ::= & \ldots \mid e.f \\ d & ::= & \ldots \mid d.f \end{array}$$

Define during elaboration:

$$e{\rightarrow}f \equiv (*e).f$$

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \ldots \mid \textsf{struct}\ s$$

Extend expressions with field access:

$$\begin{aligned} e &::= \ \ldots \mid e.f \\ d &::= \ \ldots \mid d.f \end{aligned}$$

Define during elaboration:

$$e{\rightarrow}f \equiv (*e).f$$

Type rule:

$$\frac{\Gamma \vdash e : \textsf{struct}\ s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \dots \mid \text{struct } s$$

Extend expressions with field access:

$$\begin{aligned} e &::= \dots \mid e.f \\ d &::= \dots \mid d.f \end{aligned}$$

Define during elaboration:

$$e \rightarrow f \equiv (*e).f$$

Can also use this as a destination

Type rule:

$$\frac{\Gamma \vdash e : \text{struct } s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Expressions and Typing

Extend types with struct types:

$$\tau ::= \ \ldots \mid \textsf{struct}\ s$$

Extend expressions with field access:

$$\begin{aligned} e &::= \ldots \mid e.f \\ d &::= \ldots \mid d.f \end{aligned}$$

Define during elaboration:

$$e{\rightarrow}f \equiv (*e).f$$

Can also use this as a destination

Type rule:

Struct s must have been defined.

$$\frac{\Gamma \vdash e : \textsf{struct}\ s \qquad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Expressions and Typing

How to even get an expression of type struct s?

Extend types with struct types:

$$\tau ::= \ \dots \mid \textsf{struct}\ s$$

Extend expressions with field access:

$$\begin{aligned} e &::= \dots \mid e.f \\ d &::= \dots \mid d.f \end{aligned}$$

Define during elaboration:

$$e{\rightarrow}f \equiv (*e).f$$

Can also use this as a destination

Type rule:

Struct s must have been defined.

$$\frac{\Gamma \vdash e : \textsf{struct}\ s \quad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Expressions and Typing

How to even get an expression of type struct s?

Extend types with struct types:

$$\tau ::= \ \ldots \mid \textsf{struct}\ s$$

```
alloc(struct s)
```

Extend expressions with field access:

$$\begin{array}{lll} e & ::= & \ldots \mid e.f \\ d & ::= & \ldots \mid d.f \end{array}$$

Define during elaboration:

$$e{\rightarrow}f \equiv (*e).f$$

Can also use this as a destination

Type rule:

Struct s must have been defined.

$$\frac{\Gamma \vdash e : \textsf{struct}\ s \qquad s.f : \tau}{\Gamma \vdash e.f : \tau}$$

Dynamic Semantics

Example

Consider the following program fragment:

```
struct point {
  int x;
  int y;
};

struct point* p = alloc(struct point);
```

How should the following expressions evaluated?

$$(*p).y$$

Example

Consider the following program fragment:

```
struct point {
  int x;
  int y;
};

struct point* p = alloc(struct point);
```

Fields are filled with default values.

How should the following expressions evaluated?

$(*p).y$

Evaluation of Structs ($(*p).y$)

Option: Evaluate the struct first

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

$$H\ ;\ S\ ;\ \eta \vdash \{x = v_1, y = v_2\} \rhd (\_.y\ ,\ K) \qquad \longrightarrow \qquad H\ ;\ S\ ;\ \eta \vdash v_2 \rhd K$$

Evaluation of Structs ($(*p).y$)

Option: Evaluate the struct first

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

$$H\ ;S\ ;\eta \vdash \{x = v_1, y = v_2\} \rhd (\_.y\ ,\ K) \qquad \longrightarrow \qquad H\ ;S\ ;\eta \vdash v_2 \rhd K$$

- But how would we implement that?
- We cannot store the struct in a register

Evaluation of Structs ($(*p).y$)

Option: Evaluate the struct first

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

$$H \; ; S \; ; \eta \vdash \{x = v_1, y = v_2\} \rhd (\_.y \; , K) \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash v_2 \rhd K$$

- But how would we implement that?
- We cannot store the struct in a register

Better:

- First get the address of struct p
- Take the field offset of y (4 bytes in this case)
- Retrieve integer at address p+4

‘Address Of’ Operator

In C we can get the address of a variable x using and a field f

```
&((*p).f)        &x
```

‘Address Of’ Operator

In C we can get the address of a variable x using and a field f

```
&((*p).f)        &x
```

x has to be stored on the stack.

‘Address Of’ Operator

In C we can get the address of a variable x using and a field f

```
&((*p).f)      &x
```

x has to be stored on the stack.

- In C0 we cannot take the address of variables
- This would complicated the semantics
- However, we will use the ‘address of’ operator in the semantics

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Rules:

$$H \; ; S \; ; \eta \vdash e.f \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash *(\&(e.f)) \rhd K$$

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Rules:

Get the address of e.f

$$H \; ; S \; ; \eta \vdash e.f \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash *(\&(e.f)) \rhd K$$

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Rules:

Get the address of e.f

$$H \; ; S \; ; \eta \vdash e.f \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash *(\&(e.f)) \rhd K$$

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

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Rules:

Get the address of e.f

$$H \; ; S \; ; \eta \vdash e.f \rhd K \quad \longrightarrow \quad H \; ; S \; ; \eta \vdash *(\&(e.f)) \rhd K$$

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

$$H \; ; S \; ; \eta \vdash a \rhd (\&(\_.f) \; , K) \quad \longrightarrow \quad H \; ; S \; ; \eta \vdash a + \mathsf{offset}(s, f) \rhd K$$
$$(a \neq 0, a : \mathsf{struct}\; s)$$

Evaluation of Field Access

- We expression `e` has a large type, we evaluate `*e` by evaluating `e` to an address but we don't dereference it

- This is similar to a destination `*d` on the left-hand side of an assignment

Get the address of e.f

Rules:

$$H\ ;S\ ;\eta \vdash e.f \rhd K \quad \longrightarrow \quad H\ ;S\ ;\eta \vdash *(\&(e.f)) \rhd K$$

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

$$H\ ;S\ ;\eta \vdash a \rhd (\&(\_.f)\ ,\ K) \quad \longrightarrow \quad H\ ;S\ ;\eta \vdash a + \mathsf{offset}(s, f) \rhd K \quad (a \neq 0, a : \mathsf{struct}\ s)$$

$$H\ ;S\ ;\eta \vdash a \rhd (\&(\_.f)\ ,\ K) \quad \longrightarrow \quad \mathsf{exception(mem)} \quad (a = 0)$$

Evaluation Rules

Evaluation Rules

$$H \;;\; S \;;\; \eta \vdash \texttt{\&}(*e) \triangleright K \qquad \longrightarrow \qquad H \;;\; S \;;\; \eta \vdash e \triangleright K$$

Evaluation Rules

$$H\ ;\ S\ ;\ \eta \vdash \texttt{\&}(*e) \triangleright K \quad \longrightarrow \quad H\ ;\ S\ ;\ \eta \vdash e \triangleright K$$

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

Evaluation Rules

$$H \; ; S \; ; \eta \vdash \texttt{\&}(*e) \rhd K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash e \rhd K$$

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

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

Evaluation Rules

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

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

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

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

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

Evaluation Rules

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

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

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

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

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

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

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

Evaluation Rules

These are the only cases in which we can get a large type: field deref, pointer deref, and array access.

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

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

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

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

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

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

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

Example: Iteration of Address Calculations

```
struct point {
  int x;
  int y;
};
struct line {
  struct point A;
  struct point B;
};


struct line* L = alloc(struct line);
...
int x = (*L).B.y;
```

Example: Iteration of Address Calculations

```
struct point {
  int x;
  int y;
};
struct line {
  struct point A;
  struct point B;
};
```

```
struct line* L = alloc(struct line);
...
int x = (*L).B.y;
```

Have to compute the address of y.

Example: Iteration of Address Calculations

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, (*L).B.y) \quad \blacktriangleright\ K$$

Example: Iteration of Address Calculations

$$
\begin{array}{rl}
& H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \quad \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \quad \triangleright (\mathsf{assign}(x, \_) \; , K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, (*L).B.y) \ \ \blacktriangleright K \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash ((*L).B.y) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash *(\&((*L).B.y)) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H \ ; S \ ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \ \ \blacktriangleright K \\
\longrightarrow & H \ ; S \ ; \eta \vdash ((*L).B.y) \ \ \triangleright (\mathsf{assign}(x, \_) \ , K) \\
\longrightarrow & H \ ; S \ ; \eta \vdash *(\&((*L).B.y)) \ \ \triangleright (\mathsf{assign}(x, \_) \ , K) \\
\longrightarrow & H \ ; S \ ; \eta \vdash \&((*L).B.y) \ \ \triangleright (*(\_) \ , \mathsf{assign}(x, \_) \ , K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{rl}
 & H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, (*L).B.y) \ \ \blacktriangleright K \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash ((*L).B.y) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash *(\&((*L).B.y)) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash \&((*L).B.y) \ \ \triangleright (*(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash \&((*L).B) \ \ \triangleright (\&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{rl}
 & H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \;\; \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \;\; \triangleright (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \;\; \triangleright (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \;\; \triangleright (*(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \;\; \triangleright (\&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \;\; \triangleright (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
& H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \;\; \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \;\; \rhd (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \;\; \rhd (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \;\; \rhd (*(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \;\; \rhd (\&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \;\; \rhd (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash L \;\; \rhd (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \quad \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \quad \triangleright (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \quad \triangleright (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \quad \triangleright (*(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \quad \triangleright (\&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \quad \triangleright (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash L \quad \triangleright (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a \quad \triangleright (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
 & \qquad\qquad (\textit{given that } H \; ; S \; ; \eta(L) = a, a \neq 0)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \quad \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \quad \rhd (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \quad \rhd (\mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \quad \rhd (*(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \quad \rhd (\&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \quad \rhd (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash L \quad \rhd (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a \quad \rhd (\&(\_.B) \; , \&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
 & \hfill (\textit{given that } H \; ; S \; ; \eta(L) = a, a \neq 0) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a + 8 \quad \rhd (\&(\_.y) \; , *(\_) \; , \mathsf{assign}(x, \_) \; , K) \\
 & \hfill (\textit{given that } \mathsf{offset}(\mathsf{line}, B) = 8)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \quad \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \quad \rhd (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \quad \rhd (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \quad \rhd (*(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \quad \rhd (\&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \quad \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash L \quad \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a \quad \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
 & \hfill (\textit{given that } H \; ; S \; ; \eta(L) = a, a \neq 0) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a + 8 \quad \rhd (\&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
 & \hfill (\textit{given that } \mathsf{offset}(\mathsf{line}, B) = 8) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a + 12 \quad \rhd *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K \\
 & \hfill (\textit{given that } \mathsf{offset}(\mathsf{point}, y) = 4)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{rl}
& H \; ; S \; ; \eta \vdash \mathsf{assign}(x, (*L).B.y) \;\; \blacktriangleright K \\
\longrightarrow & H \; ; S \; ; \eta \vdash ((*L).B.y) \;\; \rhd (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash *(\&((*L).B.y)) \;\; \rhd (\mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B.y) \;\; \rhd (*(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&((*L).B) \;\; \rhd (\&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash \&(*L) \;\; \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash L \;\; \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a \;\; \rhd (\&(\_.B) \;,\; \&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
& \hfill (\textit{given that } H \; ; S \; ; \eta(L) = a, a \neq 0) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a + 8 \;\; \rhd (\&(\_.y) \;,\; *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K) \\
& \hfill (\textit{given that } \mathsf{offset}(\mathsf{line}, B) = 8) \\
\longrightarrow & H \; ; S \; ; \eta \vdash a + 12 \;\; \rhd *(\_) \;,\; \mathsf{assign}(x, \_) \;,\; K \\
& \hfill (\textit{given that } \mathsf{offset}(\mathsf{point}, y) = 4) \\
\longrightarrow & H \; ; S \; ; \eta \vdash c \;\; \rhd (\mathsf{assign}(x, \_) \;,\; K) \\
& \hfill (\textit{given that } H(a + 12) = c)
\end{array}
$$

Example: Iteration of Address Calculations

$$
\begin{array}{ll}
 & H\ ;\ S\ ;\ \eta \vdash \mathsf{assign}(x, (*L).B.y) \ \ \blacktriangleright K \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash ((*L).B.y) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash *(\&((*L).B.y)) \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash \&((*L).B.y) \ \ \triangleright (*(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash \&((*L).B) \ \ \triangleright (\&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash \&(*L) \ \ \triangleright (\&(\_.B)\ ,\ \&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash L \ \ \triangleright (\&(\_.B)\ ,\ \&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash a \ \ \triangleright (\&(\_.B)\ ,\ \&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
 & \hfill (\textit{given that } H\ ;\ S\ ;\ \eta(L) = a, a \neq 0) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash a + 8 \ \ \triangleright (\&(\_.y)\ ,\ *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K) \\
 & \hfill (\textit{given that } \mathsf{offset}(\mathsf{line}, B) = 8) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash a + 12 \ \ \triangleright *(\_)\ ,\ \mathsf{assign}(x, \_)\ ,\ K \\
 & \hfill (\textit{given that } \mathsf{offset}(\mathsf{point}, y) = 4) \\
\longrightarrow & H\ ;\ S\ ;\ \eta \vdash c \ \ \triangleright (\mathsf{assign}(x, \_)\ ,\ K) \\
 & \hfill (\textit{given that } H(a + 12) = c) \\
\longrightarrow & H\ ;\ S\ ;\ \eta[x \mapsto c] \vdash \mathsf{nop} \ \ \blacktriangleright K
\end{array}
$$

Revisiting Assignment

We can simplify the rules for assignments:

Revisiting Assignment

We can simplify the rules for assignments:

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

Revisiting Assignment

We can simplify the rules for assignments:

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

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

Revisiting Assignment

We can simplify the rules for assignments:

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

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

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

Revisiting Assignment

We can simplify the rules for assignments:

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

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

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

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

Revisiting Assignment

We can simplify the rules for assignments:

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

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

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

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

Rules for variable assignments are unchanged.

Revisiting Short-Cut Assignments

Consider statements like `d += e` again

Revisiting Short-Cut Assignments

Consider statements like `d += e` again

- If `d` is a variable `x` then we can elaborate to `assign(x,x+e)`
- If `d` denotes an address then we need to evaluate `d` first

Revisiting Short-Cut Assignments

Consider statements like `d += e` again

- If `d` is a variable `x` then we can elaborate to `assign(x,x+e)`

- If `d` denotes an address then we need to evaluate `d` first

Elaborate $d \odot= e$ to $\mathsf{asnop}(d, \odot, e)$ if $d \neq x$,

Revisiting Short-Cut Assignments

Consider statements like `d += e` again

- If `d` is a variable `x` then we can elaborate to `assign(x,x+e)`
- If `d` denotes an address then we need to evaluate `d` first

Elaborate $d \odot = e$ to $\mathsf{asnop}(d, \odot, e)$ if $d \neq x$,

$$H \; ; S \; ; \eta \vdash \mathsf{asnop}(d, \odot, e) \blacktriangleright K \qquad \longrightarrow \qquad H \; ; S \; ; \eta \vdash \&d \rhd (\mathsf{asnop}(\_, \odot, e) \; , K)$$

Revisiting Short-Cut Assignments

Consider statements like `d += e` again

- If `d` is a variable `x` then we can elaborate to `assign(x,x+e)`
- If `d` denotes an address then we need to evaluate `d` first

Elaborate $d \odot= e$ to $\mathsf{asnop}(d, \odot, e)$ if $d \neq x$,

$$H\ ;\ S\ ;\ \eta \vdash \mathsf{asnop}(d, \odot, e) \blacktriangleright K \quad \longrightarrow \quad H\ ;\ S\ ;\ \eta \vdash \&d \rhd (\mathsf{asnop}(\_, \odot, e)\ ,\ K)$$

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

Data Sizes

L4 type		size in bytes	C type
$\lvert \mathsf{int} \rvert$	$=$	4	`int`
$\lvert \mathsf{bool} \rvert$	$=$	4	`int`
$\lvert \tau * \rvert$	$=$	8	`t *`
$\lvert \tau[\,] \rvert$	$=$	8	`t *`
$\lvert \mathsf{struct}\ s \rvert$	$=$	$\mathsf{size}(s)$	`struct s`

Data Sizes

C0 and C bools have size 1 byte.

L4 type		size in bytes	C type
$\lvert \mathsf{int} \rvert$	$=$	4	`int`
$\lvert \mathsf{bool} \rvert$	$=$	4	`int`
$\lvert \tau * \rvert$	$=$	8	`t *`
$\lvert \tau[\,] \rvert$	$=$	8	`t *`
$\lvert \mathsf{struct}\ s \rvert$	$=$	$\mathsf{size}(s)$	`struct s`

Data Sizes

C0 and C bools have size 1 byte.

L4 type		size in bytes	C type
$\lvert$int$\rvert$	$=$	4	`int`
$\lvert$bool$\rvert$	$=$	4	`int`
$\lvert\tau *\rvert$	$=$	8	`t *`
$\lvert\tau[\,]\rvert$	$=$	8	`t *`
$\lvert$struct $s\rvert$	$=$	size(s)	`struct s`

- Struct sizes are determined by laying out the fields left to right
- Ints and bools are aligned at 0 modulo 4
- Pointers are aligned at 0 modulo 8
- Structs are aligned according to their most restrictive fields

Register Sizes

- With different seizes you need to maintain more information
- Need to pick the right instructions (movl vs movq)
- Need to allocate right amount of heap or stack space

▸ **Maintain size information in IRs!**

Disallow:

$$d^{64} \leftarrow s^{32}$$

Instead use:

$$\begin{aligned} d^{64} &\leftarrow \textsf{zeroextend } s^{32} \\ d^{64} &\leftarrow \textsf{signextend } s^{32} \end{aligned}$$