15-411: Structs

Jan Hoffmann

Recap: Pointers and Arrays

Pointers and Arrays: Syntax

Types

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

Expressions

$$e ::= \ldots \mid \mathsf{alloc}(\tau) \mid *e \mid \mathsf{null} \mid \mathsf{alloc\_array}(\tau, e) \mid e_1[e_2]$$

Destinations (l-values)

$$d ::= x \mid *d \mid d[e]$$

Assignments

$$\mathsf{assign}(d, e)$$

Pointers and Arrays: Static Semantics

Type rules:

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

$$\frac{\Gamma \vdash e : \tau * \quad \Gamma \nvdash e : \mathit{any}\,*}{\Gamma \vdash *e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau[\,] \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1[e_2] : \tau}$$

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

Pointers and Arrays: Dynamic Semantics

Transition system that steps between machine states

Machine states (expressions):

$$\overset{\text{Heap memory}}{H} \; ; \overset{\text{Call stack}}{S} \; ; \underset{\text{Variable environment}}{\eta} \vdash e \rhd \overset{\text{Continuation}}{K}$$

Machine states (statements):

$$H \; ; S \; ; \eta \vdash s \blacktriangleright K$$

Recap: Rules for Assignments

Variables:

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

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

Memory destinations:

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

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

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

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

Recap: Array Assignment

length(a) = H(a-8)

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

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

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

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

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

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

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

Structs

Struct Declarations and Definitions

Declaring structs:

$$\text{struct } \underbrace{s}_{\text{Type}};$$

Defining structs:

$$\text{struct } s\ \{\underbrace{\tau_1}_{\text{Type}}\ \overbrace{f_1}^{\text{Field}}; \ldots \tau_n\ f_n;\};$$

During type derivation we write the following to indicate that field f_i has type τ_i in the definition of s:

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

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

- Structs are large types that do not fit in registers

Small types:

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

Large types:

struct s

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 I

- 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 II

- 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)`
  - definitions of structs if structs are types of fields
- Struct declarations are optional (but encouraged as good style)
  - An occurrence of struct s in a context where its size is irrelevant serves as an implicit declaration of the type struct s.

Expressions and Typing

How to create an expression of type struct s?

Extend types with struct types:

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

struct s* x = `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.

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

Dynamic Semantics

Dynamics of Structs: 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 Field Access

Option: Evaluate the struct first

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

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

- This is mathematically correct but how would we implement that?
- We again give a more low-level version

Reflect efficient implementation:

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

Type Information and Field Offset

- Like for arrays, we need type information to compute the memory offset of a field
- One way to make the type information available in the dynamics is to annotate each field access in the code with the type of the struct (like we did for array access)

$$e\{\tau_1\ f_1; \ldots \tau_n\ f_n;\}.f$$

- Here, e has type struct s, which is defined by $\texttt{struct}\ s\ \{\tau_1\ f_1; \ldots \tau_n\ f_n;\}$
- The following evaluation rules omit this type information to improve readability

‘Address Of’ Operator

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

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

x has to be stored on the stack.

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

Evaluation of Field Access

- If 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

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

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

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

Type info needed

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

Evaluation of Address Operator

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

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

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

Type info omitted.

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

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

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

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

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

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;
```

Have to compute the address of y.

Example: Iteration of Address Calculations

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

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

- 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 $d \neq x$,

Want:

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

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

Not compatible with the reference C0 compiler.

Behavior of the C0 Reference Compiler

```
int main() {
  int* x = NULL;
  *x += 1 / 0;
  return 0;
}
```

Results in arithmetic exception.

Behavior of the C0 Reference Compiler

```
int f(int *x) {
  *x = 100;
  return 1;
}

int main() {
  int x = alloc(int);
  *x += f(x);
  return *x;
}
```

Returns 101.

Revisiting Short-Cut Assignments

Correct rules:

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

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

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

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

Data Sizes

C0 and C bools have size 1 byte.

L4 type		size in bytes	C type
$\lvert\text{int}\rvert$	$=$	4	`int`
$\lvert\text{bool}\rvert$	$=$	4	`int`
$\lvert\tau*\rvert$	$=$	8	`t *`
$\lvert\tau[\,]\rvert$	$=$	8	`t *`
$\lvert\text{struct } s\rvert$	$=$	$\text{size}(s)$	`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, cmpl vs cmpq)*
- Should to allocate right amount of heap or stack space

You could always use 8 bytes for spilling.

- ‣ **Maintain size information in IRs!**
- It is a good idea to keep temp/registers of different sizes separate
- If you want moves from small to large temps then make conversion explicit

Disallow:

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

Instead use:

$$\begin{array}{lcl} d^{64} & \leftarrow & \text{zeroextend } s^{32} \\ d^{64} & \leftarrow & \text{signextend } s^{32} \end{array}$$