Lecture Notes on
Structs

15-411: Compiler Design
Frank Pfenning

Lecture 15
October 14, 2014

1 Introduction

Pointers allow access to data stored in the heap. Arrays allow us to aggregate data of the same type. Structs provides means to aggregate data of different types. This creates few additional challenges in the C0 language definition and also in its implementation (and, of course, the language fragment L4 used in this course).

2 Struct Declarations and Definitions

C0 (and L4) support a subset of the struct-related constructs in C. Structs may be *declared* with

$$\mathsf{struct}\ s;$$

or they can be *defined* by specifying the fields $f_1, \ldots, f_n$ of the struct with their types

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

We will elaborate this into a form where, for typing purposes, we know $s.f_i : \tau_i$. For compilation purposes we also compute $\mathsf{offset}(s, f_i)$; see remarks later in this lecture.

Because structs might require an arbitrary amount of memory, we stipulate that they can never be held in variables, but must be allocated on the heap. To specify this concisely we distinguish *small types* from *large types*. Values of small type fit in registers, while values of large type must be on the heap. In L4, we have

- small types int, bool, $\tau*$, $\tau[\,]$, and
- large types $\mathsf{struct}\ s$

We have the following significant restrictions on 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.

There are some scoping requirements imposed on structs, but they are surprisingly lenient. The reason is that undefined structs provide a very weak form of polymorphism. For example, we can pass values of type struct $s\ *$ as pointers without needing to know how struct s is defined, as long as we don't attempt to access its fields. The following static semantic rules apply:

1. Field names occupy their own name space, so they cannot clash with variable, function, or type names (but they must be distinct from keywords). The same field names can be used in different struct definitions.
2. In a given struct definition, all field names must be distinct.
3. A struct may be defined at most once.
4. Types struct s that have not (yet) been defined may be referenced as long as their size is irrelevant. The size of a struct is *relevant* in expressions alloc(struct s), alloc_array(struct s, e), and in struct definitions when serving as the type of a field.
5. 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).

3 Expressions and Typing

The extension of the language of expressions and destinations is surprisingly economical.

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

We also define (typically during elaboration):

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

which can also be used as a destination in the form $d{\rightarrow}f$.

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

For this rule to apply, struct s must have been defined. It is not sufficient for it to have just been declared, because we could not determine the type of field f.

Because destinations are also expressions, no additional typing rules are needed for destinations. But recall from the restrictions in Section 2 that prior rules are severely restricted by allowing only small types.

4 Dynamic Semantics

As might be suspected, the dynamic semantics for structs is more difficult. This is because we write programs as if structs would fit into variables; in reality we are mostly manipulating their addresses. For example, under the definition

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

and after

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

the expression $(*p).y$ should evaluate to 0. But what is the value of $*p$? We cannot just dereference the p, since that just holds the *address* of the beginning of the struct stored in memory. Instead we have to use this address itself and then compute the offset of the field y (4, under the x86-64 ABI we are using), counting from the beginning of the struct, add that to p, and then retrieve the integer stored in that position.

So, in this context (when the expression has a large type), we evaluate $*e$, essentially just taking the value of e but not dereferencing it. This is quite similar to what we have to do when $*d$ appears as an l-value on the left-hand side of an assignment. To unify these, we introduce the new construct $\texttt{\&}e$ where e has a large type. This is not an extension of the source language (which would greatly complicate its semantics), but we use it in the description of the operational semantics.

It first use in field access. For an expression $e.f$ we compute the address of e (rather than its "value" which cannot be directly represented).

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

Next we have several rules for computing addresses of expressions of large type.

$$
\begin{array}{lcl}
H \; ; S \; ; \eta \vdash \&(*e) \rhd K & \longrightarrow & H \; ; S \; ; \eta \vdash e \rhd K \\[1em]
H \; ; S \; ; \eta \vdash \&(e.f) \rhd K & \longrightarrow & H \; ; S \; ; \eta \vdash \&e \rhd (\&(\_.f) \; , K) \\
H \; ; S \; ; \eta \vdash a \rhd (\&(\_.f) \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash a + \mathsf{offset}(s, f) \rhd K \\
 & & \qquad\qquad (a \neq 0, a : \mathsf{struct}\; s) \\
H \; ; S \; ; \eta \vdash a \rhd (\&(\_.f) \; , K) & \longrightarrow & \mathsf{exception}(\mathsf{mem}) \qquad (a = 0) \\[1em]
H \; ; S \; ; \eta \vdash \&(e_1[e_2]) \rhd K & \longrightarrow & H \; ; S \; ; \eta \vdash e_1 \rhd (\&(\_[e_2]) \; , K) \\
H \; ; S \; ; \eta \vdash a \rhd (\&(\_[e_2]) \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash e_2 \rhd (\&(a[\_] \; , K) \\
H \; ; S \; ; \eta \vdash i \rhd (\&(a[\_] \; , K) & \longrightarrow & H \; ; S \; ; \eta \vdash a + i|\tau| \rhd K \\
 & & a \neq 0, 0 \leq i < \mathsf{length}(a), a : \tau[\,] \\
H \; ; S \; ; \eta \vdash i \rhd (\&(a[\_] \; , K) & \longrightarrow & \mathsf{exception}(\mathsf{mem}) \\
 & & a = 0 \text{ or } i < 0 \text{ or } i \geq \mathsf{length}(a)
\end{array}
$$

These are the only cases, because they are the only possibilities for expressions of large type: a field dereference, a pointer dereference, or an array access.

Because of the layout requirements of C and C0, address calculation sometimes have to iterate. This is captures in the rules above by one address calculus invoking another in the continuation. To see the need for that, we extend the example above.

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

In the code fragment

```
struct line* L = alloc(struct line);
...
(*L).B.y = 42;
```

we have to compute the address of $(*L).B.y$. Assuming an environment η and some continuation K, this computation proceeds as follows:

$$
\begin{array}{lll}
 & \eta \vdash \&((*L).B.y) \rhd K & \\
\longrightarrow & \eta \vdash \&((*L).B) \rhd (\&(\_.y) \; , K) & \\
\longrightarrow & \eta \vdash \&(*L) \rhd (\&(\_.B) \; , \&(\_.y) \; , K) & \\
\longrightarrow & \eta \vdash L \rhd (\&(\_.B) \; , \&(\_.y) \; , K) & \\
\longrightarrow & \eta \vdash a \rhd (\&(\_.B) \; , \&(\_.y) \; , K) & (\eta(L) = a, a \neq 0) \\
\longrightarrow & \eta \vdash a + 8 \rhd (\&(\_.y) \; , K) & (\mathsf{offset}(B, \mathsf{line}) = 8) \\
\longrightarrow & \eta \vdash a + 12 \rhd K & (\mathsf{offset}(y, \mathsf{point}) = 4)
\end{array}
$$

5 Revisiting Assignment

We can exploit this new construct to simplify rules for assignment to destinations that are not variables (that is, they denote addresses on the heap).

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

When structs are allocated in memory, all the fields are initialized with their default values. As mentioned in the previous lecture, this just means filling the memory with 0, which is what the C library function `calloc` does.

6 Dealing with Different Data Sizes

In L2 and L3 we only had integers and booleans, but in L4 we have data of different sizes. For small types, we have the following table:

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`

Note that we have decided to represent L3 booleans as integers in C, rather than as members of the type `bool` (defined as an alias to `_Bool`). This is because booleans in C, according to the x86-64 ABI, have width 1 byte and do not need to be aligned.[1] Actually, the introduction of type `bool` to C seems relatively recent, so just using type `int` to represent truth values is not inconsistent with the C philosophy. In full C0 we decided on representing C0 booleans as C booleans, since we also have characters of width 1 byte and therefore cannot avoid dealing with data of size 1.

The size of a struct type is computed by laying out the structs in memory from left to right, inserting padding to make sure that each field is properly aligned. Each integer and boolean must be aligned at 0 modulo 4, each pointer or array reference must be aligned at 0 modulo 8, and each enclosed struct must be aligned according to its most stringent field requirement. Furthermore, we add padding at the end so that the whole struct has a size which is 0 modulo its most stringent field requirement. This is so arrays can be laid out simply by knowing the size of its type. The C library function `calloc` should always return a pointer that is 0 modulo 8 and therefore appropriate for any struct we might want to allocate.

[1] This created some significant complications in writing the compiler for L3 that we wanted to avoid.

7 Detail: Register Sizes

Dealing with data of different sizes will likely require maintaining additional information in your compiler so you can pick the right load/store and register movement instructions (`movl` vs. `movq`), the right comparisons (`cmpl` vs. `cmpq`), reserve the appropriate amount of stack space, allocate the appropriate amount of heap space, and do correct address calculations.

The good news is that in L3 and L4, registers only need to hold 4 byte or 8 byte values. Still, it is very easy to introduce bugs when you do not explicitly mediate changes in data size. For example, for the intermediate form we recommend disallowing instructions of the form

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

where s and d are registers of the indicated sizes, but writing one of

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

and similarly for truncations in the other directions. This should ensure that you do not accidentally apply incorrect transformations, like copy propagation, if the destination and source of a "move" have different sizes.

On the x86-64 architecture, both move and arithmetic instructions that target a 32-bit register have the peculiar effect of zero-extending the value into the whole 64-bit register. For example,

```
MOVL %EAX, %EAX
```

has an effect: it replaces bits 32–63 of `%RAX` by 0. Similarly,

```
XORL %EAX, %EAX
```

will set all 64 bits of `%RAX` to 0, not just the lowest 32. For more on this we recommend Bryant and O'Hallaron's note on x86-64 Machine-Level Programming, in particular the information on Page 9.